Provably Sound Replay, Crash Recovery, and Noninterference for Nondeterministic LLM Agents

Aniketh Bandlamudi
University of Virginia
zpq3uh@virginia.edu me@aniketh.net

November 2026

Abstract

Agents built on large language models (LLMs) run for minutes or hours, call tools with real side effects, and crash. Because model calls are nondeterministic, re-running a crashed agent re-samples every call and can repeat work that already happened. Durable-execution engines avoid this by recording each nondeterministic result and replaying it, but they treat each result as an opaque value and say nothing about the distribution it came from. We give a small probabilistic semantics for an LLM agent in which every model call is a draw from an abstract probability measure and every response is recorded in an append-only journal, with record, replay, and resume modes. We prove that replaying a run from its journal reproduces its trace and journal exactly, so replay succeeds with probability 1 and record and replay induce the same distribution over traces. We also prove that after a crash at any point, resuming with fresh randomness behaves exactly like an uninterrupted run whose randomness is spliced at the crash point, so the continuation follows the original distribution conditioned on what survived, provided the crash does not depend on a response that was never journaled. The pointwise core of both results is mechanized in Lean 4 for every program tree and every randomness tape; the probabilistic statements follow on paper. Both results extend to systems of concurrently running agents that share memory and exchange messages, under any deterministic scheduler and with a single global journal, by a reduction to the single-agent case; the generic lifting is mechanized, and its instantiation to the multi-agent calculus is on paper. For such systems we prove a probabilistic noninterference theorem: under a floating-label discipline and a priority scheduler whose priorities depend only on public events, an agent's secret data does not influence, even statistically, what public observers see. A dependency-free Rust prototype implements the semantics, together with write-ahead logging for tool side effects. On it, 1,000 of 1,000 replays reproduce their run, and recovery from all 4,664 crash points tested completes without repeating a committed side effect. Journal appends take about 100 nanoseconds. We state precisely which restrictions these guarantees need and what lifting them requires.

Contents

1 Introduction **3**

2 Background **4**
2.1 Probabilistic Semantics 4
2.2 Durable Execution 5
2.3 Comparison 5

3 A Probabilistic Semantics with a Journal **5**
3.1 Syntax . . . 5
3.2 The Journal and Three Modes . . . 6

4 Replay Soundness and Crash Recovery **8**
4.1 Crash Recovery . . . 9
4.2 Mechanization . . . 10

5 Many Agents **11**
5.1 Systems . . . 11
5.2 Replay and Recovery for Systems . . . 12
5.3 Side Effects . . . 12
5.4 Noninterference for Concurrent Agents . . . 13

6 The Rust Prototype **15**

7 Evaluation **16**
7.1 E1: Replay Determinism . . . 17
7.2 E2: Crash Recovery at Every Journal Position . . . 17
7.3 E3: Journaling Overhead . . . 18
7.4 Threats to Validity . . . 18

8 Related Work **18**
8.1 Formal Models of LLM Agents . . . 18
8.2 Record, Replay, and Recovery . . . 19
8.3 Probabilistic Semantics and Guarantees . . . 20

9 Limitations and Future Work **20**

10 Conclusion **21**

Acknowledgments and Use of AI Tools **22**

1 Introduction

The problem

Large language models are increasingly deployed as autonomous agents that plan, call tools, and keep long-running state. Such runs fail often [6], and they are hard to make durable. A run may last minutes or hours and call external tools with real side effects, and when the process crashes part-way through, the obvious recovery, re-running the agent from the start, re-samples every model call. The new run may take a different path, and it may repeat side effects that already happened.

Durable-execution engines, exemplified by Temporal [32], solve the engineering half of this problem by recording an event history. Workflow code must be deterministic; calls to the outside world, including LLM calls, run as *Activities* whose results are recorded once and returned from the history on replay. The idea is classical: logging every nondeterministic event so that a piecewise-deterministic process can be replayed after a failure is the basis of message-logging recovery in distributed systems [11, 31]. For an LLM agent, however, this treats each model call as an opaque recorded value. The engine makes replay correct but gives no account of the distribution the value was drawn from, so it cannot say whether replay preserves the probability of a run, or what the distribution of the rest of a run is after recovery. Recent tools for LLM agents record and replay runs at their nondeterministic boundaries [8, 23] and check resume behaviour against machine-checked contracts [15, 37], but they too are deterministic or empirical.

Formal models of LLM agents, meanwhile, have focused elsewhere. LLMbda [12] gives a probabilistic, Lean-verified semantics for a single agent's program with information-flow control, and λ_A [21] and PPDL [22] give probabilistic semantics to agent programs and LLM flows. The MSC calculus [4] proves coordination properties of many agents while treating model outputs as arbitrary. None of them models journaling, replay, or crash recovery.

Contribution

We give a minimal probabilistic semantics for durable replay of an LLM agent and prove that replay and crash recovery are sound in a precise probabilistic sense.

1. **A probabilistic semantics with a journal (Section 3).** Every model call is a draw from an abstract probability measure and every response is recorded in an append-only journal; the agent runs in record, replay, or resume mode. Fixing the randomness on a tape lets runs in different modes be compared directly.

2. **Replay soundness (Theorem 4.2).** Replaying a run from its journal reproduces its trace and journal exactly. Replay therefore succeeds with probability 1, and recording then replaying induces the same distribution over traces as recording alone.

3. **Crash recovery (Corollary 4.3).** After a crash at any point, resuming with fresh randomness behaves exactly like an uninterrupted run whose randomness is spliced at the crash point. Resuming with the same randomness reproduces the original run, and resuming with fresh randomness continues it with the original distribution conditioned on what survived, provided the crash does not depend on a response that was never journaled.

4. **Many agents (Section 5).** Under any deterministic scheduler and with a single global journal, a system of concurrently running agents that share memory and exchange messages unfolds into a single program tree, so replay soundness and crash recovery carry over to whole systems (Theorem 5.2). We also prove a noninterference theorem for concurrently scheduled

agents (Theorem 5.3): under a floating-label discipline, with labels on memory cells and channels that never change, and a priority scheduler whose priorities depend only on public events, the public trace of a system is the same whatever its secret data, pointwise for every family of per-agent randomness tapes and hence in distribution.

5. **A Lean mechanization (Section 4.2).** The pointwise core of both results, and their generic lifting to any deterministic step system, are proved in Lean 4 for every program tree, randomness tape, starting cursor, and crash point, with no unproven steps; the instantiation to the multi-agent calculus and the noninterference theorem are proved on paper.

6. **A prototype and measurements (Sections 6 and 7).** A dependency-free Rust crate implements the semantics together with write-ahead logging for tool side effects. We measure replay determinism, recovery from a crash at every journal position, and journaling overhead.

The paper's source, the Lean development, and the Rust prototype with all experiment harnesses are available at `https://github.com/ankthba/agentical`.

Scope

Logging nondeterministic events for replay is not new; what is new is a semantics in which recovery with fresh randomness provably preserves the distribution of a run (Corollary 4.3(ii)), not only individual runs. Replay never repeats a side effect, but whether recovery may perform an interrupted effect again depends on the external world, which we do not formalize (Section 5.3). The multi-agent results assume deterministic scheduling and a single global journal; the prototype runs a single agent. Section 9 lists what lifting these restrictions requires.

Roadmap

Section 2 reviews probabilistic semantics and durable execution. Section 3 defines the calculus, Section 4 proves replay soundness and crash recovery, and Section 5 extends both to systems of agents and proves noninterference. Section 6 describes the prototype and Section 7 the experiments. Section 8 discusses related work, Section 9 limitations and future work, and Section 10 concludes.

2 Background

This section reviews the two traditions the paper connects: probabilistic semantics, and durable execution by logging and replay.

2.1 Probabilistic Semantics

Probabilistic program semantics goes back at least to Kozen [17], who interpreted programs as transformers of probability measures. Borgström et al. [5] give distribution-based big-step and small-step semantics for a probabilistic call-by-value λ-calculus and prove them equivalent, together with a sampling-based semantics that agrees with them when integrated over traces. Our semantics is of the sampling kind: a run reads its random choices from a tape, and the distribution over runs is induced by the distribution over tapes (Lemma 3.1). Fixing the tape is also what lets us compare runs in different modes directly, an instance of the coupling method for relational reasoning about probabilistic programs [2]. Probabilistic automata [28] add nondeterministic choice, resolved by a scheduler, to probabilistic transitions; a single agent needs no scheduler, and concurrent agents use one (Section 5).

2.2 Durable Execution

Durable-execution systems, Temporal [32] being the canonical production example, give long-running workflows crash-recoverable semantics. The mechanism is an *event history*: an append-only log of all externally determined values received by the workflow. On restart, the workflow code is re-executed from the beginning, but any call that would normally block on an external response instead returns the stored value from the log. Workflow code must therefore be deterministic: re-run against the same history, it must make the same decisions. Calls to the outside world, including LLM calls, run as *Activities* whose results are recorded once and thereafter returned from the history, and other nondeterministic values such as random numbers are obtained through replay-safe SDK operations that record them [33]. This makes replay correct, but it gives no semantics to the *distribution* of an Activity's outcome, only to the single realized value.

The technique itself is classical. Message-logging protocols for rollback recovery [11, 31] assume that processes are *piecewise deterministic*: execution is deterministic between nondeterministic events, which are logged and replayed. Our journal applies the same idea to an LLM agent, for which every model call is a nondeterministic event, and our contribution is to state and prove its soundness inside a probabilistic semantics.

2.3 Comparison

Table 1 compares this work with the closest related systems; Section 8 discusses them in detail.

Table 1: Comparison with the closest related systems. ✓ = supported, × = not supported, ∼ = partial.

System	Multi-agent	Prob. semantics	Durable replay	Mechanized	Impl.
LLMbda [12]	×	✓	×	✓ (Lean)	✓ (verified interpreter)
MSC coord. [4]	✓	× (black box)	∼ (ZipperGen, informal)	✓ (Lean 4)	✓ (Python)
Temporal [32]	∼	× (det. required)	✓	×	✓
This work	✓ (det. scheduling)	✓	✓	✓ (Lean, pointwise)	∼ (Rust, single agent)

3 A Probabilistic Semantics with a Journal

We give a small calculus for a single agent whose only source of nondeterminism is calls to a language model, together with a journal that records every outcome. The calculus is deliberately minimal: it isolates the part of an agent that durable execution has to get right, namely the interplay between stochastic model calls and the record that makes them reproducible.

3.1 Syntax

Let *Val* be a countable set of values; prompts and model outputs are values (for example, finite token sequences). Expressions e are pure and deterministic: $e \Downarrow v$ is a total function from closed expressions to values, and we assume programs are well typed, so the condition of an **if** evaluates to a boolean. Terms are

$$M, N \;::=\; \mathbf{return}\; e \;\mid\; \mathbf{let}\; x = e\; \mathbf{in}\; M \;\mid\; \mathbf{if}\; e\; \mathbf{then}\; M\; \mathbf{else}\; N \;\mid\; \mathbf{let}\; x \Leftarrow \mathbf{oracle}(e)\; \mathbf{in}\; M.$$

The term $\mathbf{let}\; x \Leftarrow \mathbf{oracle}(e)\; \mathbf{in}\; M$ asks the model with the prompt that e evaluates to and binds the response to x in M. An *oracle* assigns to each prompt p a probability measure μ_p on *Val*; since *Val* is countable, μ_p is discrete. Nothing else is assumed about μ_p: the results below hold for every

oracle. The same construct models any externally determined value, such as a tool response, a message from the environment, or a clock read; we speak of model calls for concreteness. We omit general recursion, so every run makes finitely many oracle calls (Section 9 discusses lifting this).

Programs as trees. Because expressions are pure, the behaviour of a closed term is captured by its tree of oracle calls: either it halts with a value, or it makes a call with some prompt p and continues with a subtree that depends on the response. Formally, define $T(M)$ by

$$\begin{aligned}
T(\mathbf{return}\ e) &= \mathsf{halt}(v) && \text{where } e \Downarrow v,\\
T(\mathbf{let}\ x = e\ \mathbf{in}\ M) &= T(M[v/x]) && \text{where } e \Downarrow v,\\
T(\mathbf{if}\ e\ \mathbf{then}\ M\ \mathbf{else}\ N) &= T(M) \text{ or } T(N) && \text{as } e \Downarrow \mathsf{true} \text{ or } \mathsf{false},\\
T(\mathbf{let}\ x \Leftarrow \mathbf{oracle}(e)\ \mathbf{in}\ M) &= \mathsf{draw}\big(p,\ v \mapsto T(M[v/x])\big) && \text{where } e \Downarrow p.
\end{aligned}$$

Trees are well founded because terms have no recursion. All definitions and results below depend only on $T(M)$, and the Lean development (Section 4.2) works directly with these trees.

3.2 The Journal and Three Modes

A *journal* J is a finite list of entries (c, p, v): the cursor c (the entry's position in the log), the prompt p, and the response v. The agent runs in one of three modes.

Record. Each call is answered by the oracle, and its entry is appended to the journal.
Replay. Each call is answered from the next journal entry; the oracle is not consulted.
Resume. Replay until the journal is exhausted, then switch to record mode for the rest of the run. This is recovery after a crash.

A run emits a *trace*: one label $o(c, p, v)$ per call, recording its cursor, prompt, and response, followed by $\mathsf{done}(v)$ when the run halts with v.

Randomness tapes. To compare runs in different modes we fix their randomness in advance. A *tape* is a function $\omega : \mathbb{N} \times \mathit{Val} \to \mathit{Val}$, where $\omega(c, p)$ is the response the oracle gives to prompt p at cursor c. Let Ω be the set of tapes with the product measure $\mathbb{P} = \bigotimes_{(c,p)} \mu_p$, under which the coordinates are independent and $\omega(c, p)$ has law μ_p. Since Val is countable, so is the set of finite traces, so every set of traces is measurable and every function of a run defined below is a random variable.

Semantics. Write $\mathsf{rec}(T, \omega, c)$, $\mathsf{rep}(T, J)$, and $\mathsf{res}(T, J, \omega, c)$ for the (trace, journal) pair produced by running tree T from cursor c in each mode:

$$\begin{aligned}
\mathsf{rec}(\mathsf{halt}(v), \omega, c) &= (\mathsf{done}(v),\ \varepsilon)\\
\mathsf{rec}(\mathsf{draw}(p, k), \omega, c) &= o(c, p, v) \cdot \mathsf{rec}(k(v), \omega, c{+}1) && v = \omega(c, p),\ \text{entry } (c, p, v)\\
\mathsf{rep}(\mathsf{halt}(v), J) &= (\mathsf{done}(v),\ \varepsilon)\\
\mathsf{rep}(\mathsf{draw}(p, k), (c', p', v) \cdot J) &= o(c', p, v) \cdot \mathsf{rep}(k(v), J) && \text{entry } (c', p, v)\\
\mathsf{rep}(\mathsf{draw}(p, k), \varepsilon) &= (\varepsilon, \varepsilon) && \text{journal exhausted}
\end{aligned}$$

Here $\ell \cdot (t, J)$ prepends the label ℓ to the trace t and the indicated entry to J; replay re-emits each journal entry it consumes, so its output journal can be compared with the recorded one. Resume

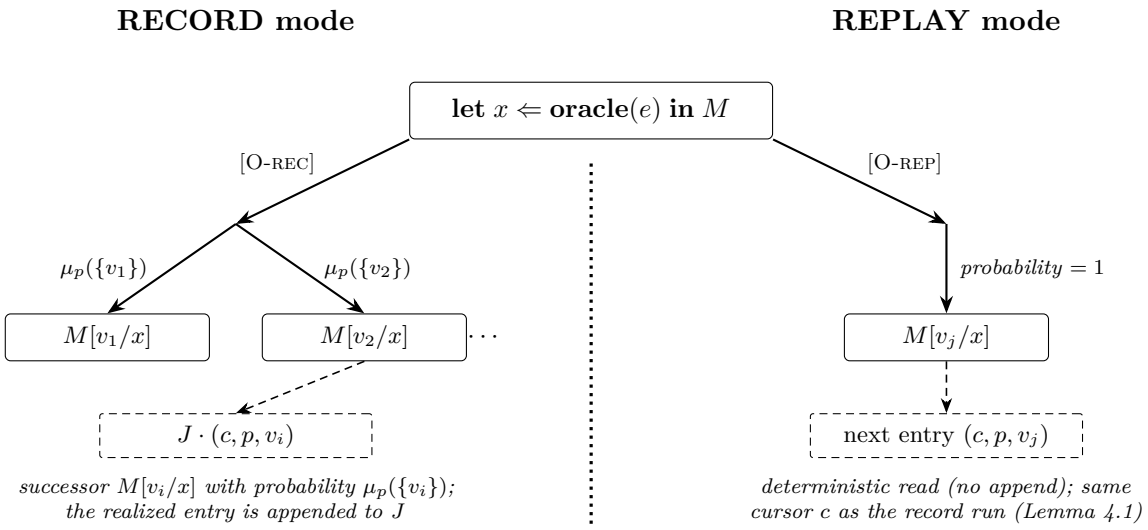


Figure 1: Record and replay of one oracle call. In record mode (left) the call samples $v \sim \mu_p$ and appends (c, p, v) to the journal. In replay mode (right) the same call reads the next entry with probability 1 and appends nothing new. The cursor c does not depend on any sampled value (Lemma 4.1), so both modes address the same entry.

mode replays while the journal lasts and then records, tracking the cursor c so that the first fresh call lands at the position after the last surviving entry:

$$\begin{aligned}
\mathsf{res}(\mathsf{halt}(v), J, \omega, c) &= (\mathsf{done}(v),\ \varepsilon) \\
\mathsf{res}(\mathsf{draw}(p,k), (c', p', v) \cdot J, \omega, c) &= o(c', p, v) \cdot \mathsf{res}(k(v), J, \omega, c{+}1) && \text{entry } (c', p, v) \\
\mathsf{res}(\mathsf{draw}(p,k), \varepsilon, \omega, c) &= o(c, p, v) \cdot \mathsf{res}(k(v), \varepsilon, \omega, c{+}1) && v = \omega(c, p),\ \text{entry } (c, p, v)
\end{aligned}$$

These functions correspond to the small-step rules

$$\begin{aligned}
&[\text{O-REC}] && \langle \mathsf{draw}(p,k), J, c\rangle \xrightarrow{o(c,p,v)} \langle k(v), J \cdot (c,p,v), c{+}1\rangle && \text{with probability } \mu_p(\{v\}), \\
&[\text{O-REP}] && \langle \mathsf{draw}(p,k), (c',p',v) \cdot R, c\rangle \xrightarrow{o(c',p,v)} \langle k(v), R, c{+}1\rangle && \text{with probability 1}, \\
&[\text{O-RESUME}] && \langle \mathsf{draw}(p,k), \varepsilon, c\rangle \text{ in resume mode steps as [O-REC]},
\end{aligned}$$

where in replay and resume mode R is the part of the journal not yet replayed. In replay mode an exhausted journal at a call is stuck; in resume mode it falls through to recording. [O-REC] is the only rule whose successor is not a single configuration, so all randomness in a run comes from oracle calls. Replay checks neither the journaled prompt nor the cursor against the program: it trusts that it is replaying the same program from the same starting cursor (assumption A2 below).

The record distribution. The record run samples each response from the oracle as it goes. Its distribution is that of $\mathsf{rec}(T, \omega, c)$ for $\omega \sim \mathbb{P}$:

Lemma 3.1 (Tapes realize sampling)**.** *Call $t = o(c, p_1, v_1) \cdots o(c{+}n{-}1, p_n, v_n)\,\mathsf{done}(v)$ a* trace of T from c *if it follows a path of T: each p_i is the prompt T issues after the responses $v_1, \ldots, v_{i-1}$, and T then halts with v. For every tree T, cursor c, and finite trace t of this form,*

$$\mathbb{P}\big[\,\mathsf{rec}(T, \omega, c) \text{ has trace } t\,\big] \;=\; \prod_{i=1}^{n} \mu_{p_i}(\{v_i\})$$

if t is a trace of T, and 0 otherwise. This is the probability that the sampling semantics (rule [O-REC]*) produces t.*

Proof. The i-th call of the run is made at cursor $c+i-1$ with a prompt p_i that is determined by the responses $v_1, \ldots, v_{i-1}$, and its response is the coordinate $\omega(c+i-1, p_i)$. The event that the run produces t is therefore the event that $\omega(c+i-1, p_i) = v_i$ for $i = 1, \ldots, n$, where $p_1, \ldots, p_n$ are the prompts that T issues along t (fixed once t is fixed). These are n distinct coordinates of the tape, so by independence the event has probability $\prod_i \mu_{p_i}(\{v_i\})$. The sampling semantics produces t with the same probability, by the chain rule: given the first $i-1$ responses, the i-th is drawn from μ_{p_i}. □

4 Replay Soundness and Crash Recovery

We state three results: the cursors of a run do not depend on its randomness (Lemma 4.1), replay reproduces a recorded run exactly (Theorem 4.2), and resuming after a crash continues the run with the correct distribution (Corollary 4.3). Each comes in a pointwise form, quantified over every tree and every tape, which is mechanized in Lean 4 (Section 4.2), and a probabilistic form, which follows on paper from the pointwise one. Throughout we assume:

(A1) *Val* is countable and each μ_p is a probability measure on it.
(A2) Replay and resume run the same program, from the same starting cursor, as the recorded run.
(A3) The journal is not corrupted or reordered; a crash keeps a prefix of it.

Lemma 4.1 (Cursors)**.** *For every tree T, tape ω, and cursor c, the i-th entry of the journal of $\mathsf{rec}(T, \omega, c)$ has cursor $c+i-1$, whatever the responses (only the number of entries depends on them).*

Proof. By induction on T: each call appends one entry at the current cursor and advances it by one. □

Theorem 4.2 (Replay soundness)**.** *For every tree T, tape ω, and cursor c, let $(t, J) = \mathsf{rec}(T, \omega, c)$. Then*

$$\mathsf{rep}(T, J) \;=\; (t, J).$$

Consequently:

(i) ***(Trace equivalence)*** *replaying the journal reproduces the recorded trace, including every cursor, prompt, and response, and the journal itself;*
(ii) ***(Determinism)*** *replay is a function of the journal alone and reads no randomness, so given J it produces t with probability 1;*
(iii) ***(Equiprobability)*** *for every set Φ of traces,*

$$\mathbb{P}\big[\,\mathsf{rec}(T, \omega, c) \text{ has trace in } \Phi\,\big] \;=\; \mathbb{P}\big[\,\mathsf{rep}(T, J_\omega) \text{ has trace in } \Phi\,\big],$$

where J_ω is the journal of $\mathsf{rec}(T, \omega, c)$; that is, recording and then replaying induces the same distribution over traces as recording alone, and by Lemma 3.1 a trace $o(c, p_1, v_1) \cdots o(c+n-1, p_n, v_n)\,\mathsf{done}(v)$ of T has probability $\prod_i \mu_{p_i}(\{v_i\})$ under both.

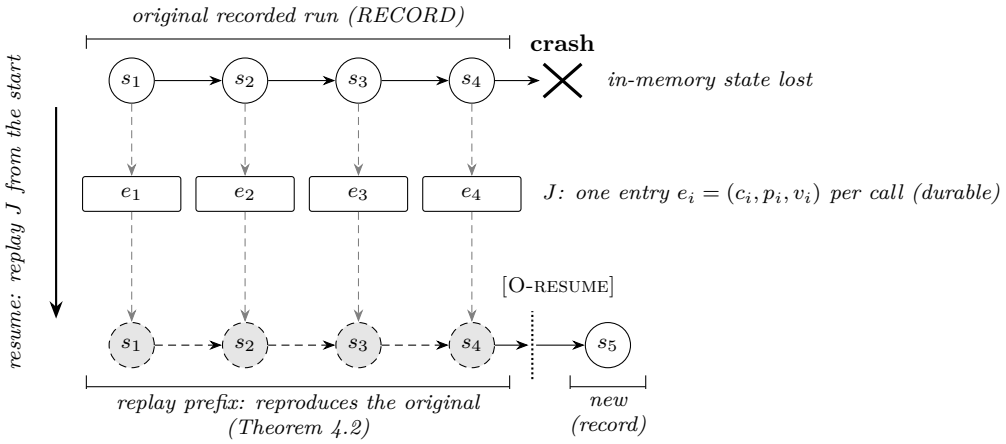


Figure 2: Crash recovery (Corollary 4.3). Calls $s_1 \dots s_4$ run in record mode and append their outcomes to the journal; a crash at s_5 loses all in-memory state but keeps the journal prefix. Resume replays the prefix with probability 1 ([O-REP]), then switches to record mode ([O-RESUME]) for the rest of the run.

Proof. The identity is proved by induction on T. For $\mathsf{halt}(v)$ both sides are $(\mathsf{done}(v), \varepsilon)$. For $\mathsf{draw}(p, k)$, recording at cursor c produces the entry (c, p, v) with $v = \omega(c, p)$ and continues as $\mathsf{rec}(k(v), \omega, c+1) = (t', J')$, so $(t, J) = (o(c, p, v)\, t',\ (c, p, v)\, J')$. Replay reads the entry (c, p, v), emits $o(c, p, v)$, and continues as $\mathsf{rep}(k(v), J')$, which equals (t', J') by the induction hypothesis for the subtree $k(v)$. Part (i) is the first component of the identity, and part (ii) holds because rep does not take a tape. For part (iii), the identity holds for every ω, so the two events are the same subset of Ω and have the same probability; the product formula is Lemma 3.1. □

Part (iii) is immediate from part (i): it holds for every law on tapes, not only $\mathbb{P}$, and the same is true of deterministic replay under any input distribution. The probabilistic content of the paper lies elsewhere: in Lemma 3.1, which says that the tape semantics realizes sampling from the oracle, and in Corollary 4.3(ii), which says that recovery with fresh randomness preserves the distribution of a run.

4.1 Crash Recovery

A crash loses the agent's in-memory state but, by (A3), keeps a prefix of the journal. Recovery resumes the program on that prefix with whatever randomness is available afterwards. To state the result, define the *splice* of two tapes at cursor m by

$$\mathsf{splice}_m(\omega, \omega')(c, p) \;=\; \begin{cases} \omega(c, p) & c < m, \\ \omega'(c, p) & c \geq m. \end{cases}$$

Corollary 4.3 (Crash recovery)**.** *For every tree T, tapes ω and ω', cursor c, and $n \in \mathbb{N}$, let J be the journal of $\mathsf{rec}(T, \omega, c)$ and $J_{<n}$ its first n entries. Then*

$$\mathsf{res}(T,\ J_{<n},\ \omega',\ c) \;=\; \mathsf{rec}\big(T,\ \mathsf{splice}_{c+n}(\omega, \omega'),\ c\big).$$

Consequently:

(i) ***(Same randomness)*** *resuming with $\omega' = \omega$ reproduces the uninterrupted run exactly, wherever the crash occurred;*

(ii) ***(Fresh randomness)*** *if ω' is drawn from $\mathbb{P}$ independently of ω, the resumed run has the same distribution as an uninterrupted record run, and conditioned on the surviving prefix $J_{<n}$, its continuation has the same distribution as the original run's continuation conditioned on the same prefix.*

Proof. The identity is proved by induction on T, generalizing over c and n, using one auxiliary fact: a record run from cursor c reads the tape only at cursors $\geq c$, so two tapes that agree from c on give the same run. If $n = 0$, resuming on the empty journal is recording on ω', and $\mathsf{splice}_c(\omega, \omega')$ agrees with ω' from c on. If $n > 0$ and $T = \mathsf{draw}(p, k)$, the first surviving entry is (c, p, v) with $v = \omega(c, p) = \mathsf{splice}_{c+n}(\omega, \omega')(c, p)$, so both sides make the same first call and continue with $k(v)$ from cursor $c + 1$ with $n - 1$ surviving entries, where the induction hypothesis applies.

For (i), $\mathsf{splice}_m(\omega, \omega) = \omega$. For (ii), the coordinates of $\mathsf{splice}_{c+n}(\omega, \omega')$ are independent with the right marginals, since those at cursors below $c + n$ come from ω, the rest come from ω', and ω and ω' are independent; so the spliced tape has law $\mathbb{P}$ and the resumed run has the record distribution. By Lemma 4.1 the surviving prefix reads only coordinates at cursors below $c + n$, on which the two tapes agree, so it is the same random variable under ω and under the spliced tape. The pair (prefix, whole run) therefore has the same joint law under both tapes, and so does the conditional law of the run given the prefix. □

Remark 4.4 (When the crash point is random)**.** Corollary 4.3 fixes the crash point n. The identity holds for every n, so it also holds when n is random. Part (ii) then still holds if the crash point is a *stopping time*, meaning whether the crash happens after k entries depends only on the coordinates at cursors below $c + k$, and is independent of ω': replacing the coordinates after a stopping time by independent fresh ones does not change the law of a tape whose cursor rows are independent and identically distributed, which is the strong Markov property. It fails if the crash depends on a response that was drawn but not yet journaled. For example, let T make one call whose response is a fair coin and return it, and let the process crash before journaling exactly when the response is 0. Then the resumed run returns 1 with probability 3/4, not 1/2. A recovery guarantee in distribution therefore needs the response to be journaled before anything that depends on it can crash the agent, which is the order [O-REC] enforces.

For a single agent every prefix of its journal is a valid recovery point, so no consistent-cut condition is needed; the same holds for systems with one global journal (Section 5.2), and Section 9 explains why per-agent journals would need one.

4.2 Mechanization

Lemma 4.1, the identity of Theorem 4.2 with part (i) and the pointwise core of part (iii), and the identity of Corollary 4.3 with part (i) are mechanized in Lean 4; part (ii) of the theorem holds by the type of the replay function, which takes no tape (`lean/AgenticalT1.lean`, checked with Lean v4.30.0). The development represents values, prompts, and cursors by natural numbers, programs by the trees of Section 3.1, and tapes by functions $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, and quantifies over every tree, tape, starting cursor, and crash point. It also proves the generic lifting of these results from program trees to any system with a deterministic step function (Section 5.2); that the multi-agent calculus is such a system is shown on paper. It contains no unproven steps, and `#print axioms` reports only Lean's standard axioms (`propext`, and `Quot.sound` for the crash-recovery theorems). The translation from terms to trees and the measure-theoretic layer, Lemma 3.1 and the distributional parts of Theorem 4.2(iii) and Corollary 4.3(ii), are on paper; their proofs use the pointwise identities, Lemma 4.1, and the independence and marginals of the tape coordinates.

5 Many Agents

We now extend the semantics to several agents that run concurrently and interact through shared memory and channels. Two results carry over: replay soundness and crash recovery hold for whole systems, by a reduction to the single-agent case (Section 5.2), and a probabilistic noninterference theorem holds for concurrently scheduled agents under a floating-label discipline (Section 5.4). Both depend on explicit restrictions, which we state as we go.

5.1 Systems

A *system* consists of agents, shared memory, and channels. An agent $id[M]_\Sigma$ has an identity, a term M, and a private state Σ that holds its local variables, its current label $pc(\Sigma)$ (Section 5.4), a count of the oracle calls it has made, and a count of the agents it has spawned. Each agent runs a single thread; parallelism comes only from running several agents. Shared memory H has a fixed set of cells, each with a fixed label ℓ_k; channels have fixed labels ℓ_c and carry first-in-first-out queues. Agent terms extend those of Section 3.1 with

$$\textbf{read}\ k\ \textbf{as}\ x\,.\,M\ \mid\ \textbf{write}\ k := e\,.\,M\ \mid\ \textbf{send}\ c\ e\,.\,M\ \mid\ \textbf{recv}\ c\ \textbf{as}\ x\,.\,M\ \mid\ \textbf{spawn}\ N\,.\,M.$$

A spawned agent's identity is (id, n), where id is the parent and n is the parent's spawn count, so identities do not depend on any response. A configuration is *well formed* if its agents' identities are distinct and no agent has an identity (x, m) with m at least the spawn count of x; steps preserve well-formedness, so identities stay unique. Every construct except an oracle call has a single successor. A receive on an empty queue is disabled, and so is any step whose side condition (Section 5.4) fails: the agent blocks rather than aborting the system. An agent whose term is **return** e halts.

A *scheduler* σ chooses, at each step, one enabled agent to step next, as a function of the run so far, or stops the system when no agent is enabled. We consider only *deterministic* schedulers, and quantify over all of them. For replay and recovery (Theorem 5.2), a randomized scheduler is covered by making its random choices oracle calls, so that they are drawn from a measure and journaled like any other; the noninterference theorem of Section 5.4 requires a deterministic scheduler.

Lemma 5.1 (Systems are programs)**.** *For every deterministic scheduler σ and initial configuration C_0, the run of the system is determined by the sequence of oracle responses: it is a tree in the sense of Section 3.1, whose calls are all the agents' oracle calls in the order the scheduler makes them.*

Proof. The run so far is a function of the responses received so far, so at each step σ picks the agent as a function of those responses, and that agent's next construct either has a single successor (pure computation, memory, channels, spawning) or is an oracle call. Unfolding the run therefore gives a tree whose internal nodes are oracle calls and whose deterministic steps are absorbed into the edges, as in the translation $T(M)$. The tree is finite in depth: every step consumes a construct of some agent's term, and a spawn splits **spawn** $N.\,M$ into N and M, so the total size of all agents' terms strictly decreases. A leaf is reached when no agent is enabled: either every agent has halted, or the remaining agents are blocked; we label the leaf with the halted agents' results in the first case and with a distinguished deadlock outcome in the second. □

The Lean development formalizes this as a *system*: a configuration type with a step function that halts, takes a deterministic step, or makes an oracle call, unfolded into a tree up to a step bound (`System.tree`). It proves that raising the bound by one never changes a run that has already halted, provided the halting output differs from the distinguished out-of-fuel output (`tree_mono`). For the calculus, a bound at least the total size of the initial terms covers the whole run.

5.2 Replay and Recovery for Systems

Let all agents share a single journal that records every oracle call, whichever agent makes it, in the order the calls occur.

Theorem 5.2 (Replay soundness and crash recovery for systems)**.** *For every deterministic scheduler and initial configuration, Theorem 4.2 and Corollary 4.3 hold for the system's run with its global journal: replaying the whole system from its initial configuration against the journal reproduces the run, and after a crash that keeps any prefix of the journal, resuming the whole system with fresh randomness gives the uninterrupted run on the spliced tape. Since every step other than an oracle call is deterministic, the run reproduced includes all memory, channel, and spawn steps, not only the oracle calls.*

Proof. By Lemma 5.1 the run is a tree, so Theorem 4.2 and Corollary 4.3 apply to it directly; in Lean these are `sys_replay_sound` and `sys_crash_resume`. Given the responses, the deterministic steps between calls are the same in record, replay, and resume, because the scheduler and every non-oracle construct are deterministic. □

The theorem's restrictions are what make it simple, and each one rules out a failure that a more ambitious design must handle.

- *One journal, totally ordered.* A crash keeps a prefix of a single log, which is automatically a consistent state. With one journal per agent, persisted independently, a crash can keep an agent's receive without the matching send, and recovery must find a consistent cut of the logs [7] that also accounts for which memory write each read observed.
- *Deterministic scheduling.* If replay could interleave agents differently from the recorded run, an agent could read a different value from shared memory than it did originally, and replay would diverge from the journal.
- *Whole-system replay.* Replay re-runs every agent from the initial configuration. Restarting one agent in the middle of a run, as a supervisor might, would also require restoring that agent's local state and suppressing messages it had already sent; we do not treat it.

5.3 Side Effects

A side effect **do** $f(e)$ changes the external world and returns a response. We model it as two journaled oracle calls: an *intent* call, whose measure is a point mass on the record (f, v) with $e \Downarrow v$, and a *response* call, whose measure is the world's response distribution for f on v. In record mode the runtime performs the effect between the two calls. Replay reads both entries from the journal and performs nothing, so the replay part of Theorem 5.2 covers effects: replay never repeats an effect. The distributional part of crash recovery does not carry over, because the world's response to an effect depends on the world's state, which is not a fixed measure per prompt.

Recovery is where effects differ from model calls. A crash between the two entries leaves an intent without a response, and resuming draws the response afresh, which in the runtime means performing the effect again; but the world may already have applied it. Whether this is safe is a property of the world, not of the journal. If f is idempotent on the world state and no other effect on the same state happened in between, performing it again leaves the world as if it had been performed once; this is the case the prototype handles by performing the effect again (Section 7.2). Otherwise recovery needs a way to ask the world whether the effect happened, or a compensating action with its own write-ahead record, and no policy that relies only on checking can guarantee exactly-once effects without a bound on how long an effect may remain in flight [20]. We do not formalize the world, so these remain assumptions rather than theorems.

5.4 Noninterference for Concurrent Agents

Agents often handle data of different sensitivity: an agent that summarizes a confidential document should not be able to leak it, even statistically, to an agent whose output is public. We give a floating-label discipline and prove that, under a restricted class of schedulers, the public behaviour of a system does not depend on its secret data.

Labels. Fix a lattice $(\mathcal{L}, \sqsubseteq)$ of security labels and a set $Lo \subsetneq \mathcal{L}$ of observable labels that is downward closed and closed under joins. Each agent carries a current label pc, which only rises. The constructs of Section 5.1 have these side conditions and label updates:

Construct	Permitted if	Then
read k	always	$pc := pc \sqcup \ell_k$
write $k := e$	$pc \sqsubseteq \ell_k$	cell k keeps its label
send $c\ e$	$pc \sqsubseteq \ell_c$	—
recv c	$pc \sqsubseteq \ell_c$	$pc := \ell_c$
oracle call with output label ℓ	$pc \sqsubseteq \ell$	$pc := \ell$
spawn N	always	the child starts with the parent's pc

A receive is a destructive read, so it requires $pc \sqsubseteq \ell_c$ like a write. Labels of cells and channels never change, and no cells or channels are created. A step whose side condition fails is disabled, so the agent blocks (Section 5.1); aborting the whole system instead would let a secret decide whether public agents run at all. For this section we exclude side effects and clock reads, which expose the external world or timing, and supervisors. The global journal of Section 5.2 records secret data, and its length reveals how much secret computation happened, so it must be readable only at the top label.

Per-agent tapes. For noninterference it is convenient to give each agent its own tape: the response to the i-th oracle call of agent id with prompt p is $\theta(id, i, p)$, for a family θ whose coordinates are independent with $\theta(id, i, p) \sim \mu_p$. Because identities are unique and each agent's call count increases, distinct calls read distinct coordinates, and the argument of Lemma 3.1 shows that this realizes sampling from the oracle, for any deterministic scheduler.

Observations. A step is *Low-visible* if the agent's current label before the step is in Lo and the data it touches is labelled in Lo: the cell for a read or write, the channel for a send or receive, and the output label for an oracle call; spawning and halting are Low-visible when the agent's label is in Lo. Low-visible steps emit labels that carry their data: $\mathsf{rd}(k, v)$, $\mathsf{wr}(k, v)$, $c!v$, $c?v$, $o(id, p, v)$, $\mathsf{spw}(id')$, and $\mathsf{done}(id, v)$. The *Low trace* tr_{Lo} of a run is its sequence of Low-visible labels; other steps are deleted, not replaced by a placeholder. An agent is *Low-active* if its label is in Lo and its next construct is not a receive on a channel whose label is outside Lo. Two configurations are *low-equivalent*, $C =_{Lo} C'$, if they have the same Low-active agents, each with the same term and private state on both sides, the same values in cells labelled in Lo, and the same queues on channels labelled in Lo. Everything else, including agents that are not Low-active, may differ.

Assumptions.

(A-SCHED) The scheduler is deterministic, and at each step it chooses the highest-priority enabled agent under a total order on agent identities that is a function of the Low trace so far. It cannot observe steps that are not Low-visible, not even how many there were.

(A-TERM) Every maximal run of consecutive steps by agents that are not Low-active is finite. In the calculus as defined every run is finite (Lemma 5.1), so this holds automatically; we state it because it is exactly what an extension with recursion would need, and it is the only termination assumption the proof uses.

(A-ORC) Each oracle response depends only on the calling agent's prompt and its own tape coordinates. This holds in the calculus by construction; in a deployment it requires that model calls at different labels share no state, such as a cache or a conversation.

Theorem 5.3 (Noninterference)**.** *Assume (A-SCHED) and (A-ORC). For every tape family θ and all well-formed configurations $C_0 =_{Lo} C_0'$, if (A-TERM) holds for the runs from C_0 and from C_0' on θ, then the two runs have the same Low trace. Consequently, for any distribution of tape families under which (A-TERM) holds almost surely for both runs, the Low traces of the two runs have the same distribution; in particular this holds for independent tapes, which realize sampling from the oracle.*

The proof rests on three lemmas.

Lemma 5.4 (Steps that are not Low-active)**.** *If $C =_{Lo} C'$ and C steps to D by an agent that is not Low-active, then $D =_{Lo} C'$, the step is not Low-visible, and it changes no Low-active agent, no cell labelled in Lo, and no channel labelled in Lo.*

Proof. Suppose first that the agent's label *pc* is not in *Lo*. Because *Lo* is downward closed, any label above *pc* is outside *Lo*, so the side conditions forbid writing a cell, sending or receiving on a channel, or making an oracle call with an output label in *Lo*; reads only raise *pc* further; and a spawned child inherits *pc* and is therefore not Low-active. The step touches only the agent itself, cells and channels outside *Lo*, and possibly a new agent that is not Low-active. Otherwise the agent's label is in *Lo* but its next construct is a receive on a channel labelled outside *Lo*; the step changes only that channel and the agent, whose label becomes $\ell_c \notin Lo$. In both cases no Low-visible label is emitted and every component that $=_{Lo}$ constrains is unchanged. □

Lemma 5.5 (Low-active steps)**.** *If $C =_{Lo} C'$ and the same Low-active agent takes its next step in both, then both steps emit the same label, or neither is Low-visible, and the resulting configurations are low-equivalent.*

Proof. The agent has the same term and private state on both sides, so it performs the same construct with the same values computed from its own state. A read of a cell in *Lo* returns the same value on both sides, and the agent's label stays in *Lo* because *Lo* is closed under joins; a read of a cell outside *Lo* may return different values, but it raises the agent's label out of *Lo* on both sides, so the agent is no longer Low-active on either side. A write or send stores the same value; if the target is in *Lo* the label is the same, and otherwise only components outside *Lo* change. A receive is on a channel in *Lo* (the agent is Low-active), so it takes the same value from the same queue. An oracle call has the same prompt and is the agent's i-th call on both sides, so by (A-ORC) it reads the same tape coordinate and gets the same response; if its output label is outside *Lo* the agent leaves the Low-active set on both sides. A spawn creates the same identity with the same term and label on both sides. Afterwards either the agent's term and state are the same on both sides, or its label is outside *Lo* on both sides; in either case it is Low-active on both sides or on neither, and the same holds for a spawned child. □

Lemma 5.6 (The next Low-active step)**.** *Let $C =_{Lo} C'$ be reached by runs with the same Low trace, and suppose (A-TERM) holds. Then either neither run takes another step by a Low-active agent, or both runs next take a step by the same Low-active agent.*

Proof. Whether a Low-active agent is enabled depends only on its term and private state (including its label, which the side conditions check) and, for a receive, on a queue of a channel in Lo, so the set E of enabled Low-active agents is the same in C and C'. By Lemma 5.4, steps by agents that are not Low-active do not change E, the Low-active agents, or the Low trace, and the scheduler's priority order depends only on the Low trace, so it is the same on both sides and stays fixed while such steps run. If E is empty, only agents that are not Low-active can step, and they can never enable a Low-active agent, so neither run takes another Low-active step. Otherwise let X be the highest-priority agent in E. On each side the scheduler fires the highest-priority enabled agent; until it fires X, it can only fire agents above X that are not Low-active, because no other Low-active agent above X is enabled. By (A-TERM) such a sequence of steps is finite, and when it ends the highest-priority enabled agent is X, which is still enabled. So both runs next take a step by X. □

Proof of Theorem 5.3. Every Low-visible step is taken by a Low-active agent: an agent whose label is outside Lo emits no Low-visible label, and a receive on a channel outside Lo is not Low-visible. By induction on the number of Low-active steps, using Lemmas 5.4 to 5.6, the two runs take the same sequence of Low-active steps with the same labels, interleaved with steps that emit no Low-visible label, and remain low-equivalent with the same Low trace after each. Hence their Low traces are equal. For the probabilistic statement, the two Low traces are equal on the event where (A-TERM) holds for both runs; if this event has probability 1, they have the same distribution. □

Each assumption rules out a known leak. Without (A-SCHED), a scheduler that counts hidden steps can turn the running time of a secret computation into a public difference [26, 34]. Without (A-TERM), a secret computation that runs forever withholds every later public event, the termination channel that termination-insensitive guarantees leave open. Without fixed cell labels, an agent holding a secret could relabel a public cell and so decide, through the secret, whether a public agent can read it; and with parallel threads inside an agent, one thread's secret receive could block another thread's public output, which is why each agent runs one thread. Finally, the discipline is coarse: an agent that has read a secret can never again produce public output, and the calculus has no construct for deliberate declassification. LLMbda [12] tracks labels on individual values for a single agent; combining its finer tracking with concurrent scheduling is open.

6 The Rust Prototype

The `agentical` Rust crate has no dependencies beyond the standard library. It defines synchronous, executor-agnostic traits for the agent, the oracle, and the journal, shown in Listing 1, along with interfaces for channels, memory, and supervision intended for future multi-agent work (Section 9).

The journal is position-indexed: the cursor is the key, as in Section 3.2, so there is no key-to-value map and no collision assumption. A runtime module implements the semantics for a single agent:

- `InMemoryJournal`, with `append`, `replay_next`, `rewind`, and `truncate` (to simulate a crash);
- `SeededOracle`, a categorical sampler driven by a SplitMix64 generator, which stands in for an LLM;
- a driver that runs an agent in record, replay, or resume mode following [O-REC], [O-REP], and [O-RESUME].

Side effects. The prototype also journals tool side effects, using a write-ahead discipline. Of this, the formal results cover only replay (Section 5.3); recovery is tested but not proved. Before

```rust
pub trait Agent {
    type State;
    type Out;
    fn step(&mut self, state: &mut Self::State) -> Step<Self::Out>;
}

pub trait Oracle {
    fn sample(&self, prompt: Prompt) -> Sampled;
    fn measure_hint(&self) -> Option<Dist> { None }
}

pub trait Journal {
    fn append(&mut self, agent: &AgentId, event: Event) -> Cursor;
    fn replay_next(&self, agent: &AgentId) -> Outcome;
    fn checkpoint(&self, agent: &AgentId) -> Cursor;
}
```

Listing 1: The agent, oracle, and journal traits (doc comments omitted).

performing an effect it appends an *intent* record carrying the effect's argument; after the effect succeeds it appends a *committed* record carrying the result. Replay returns the journaled result and does not perform the effect again. A crash between the two records leaves a dangling intent, which recovery detects and resolves either by performing the effect again with the journaled argument, which is safe for an idempotent effect of a single agent, or by recording a compensation. Replaying a compensated effect stops the run without performing anything. Section 7.2 tests this behaviour at every crash point; formalizing recovery is future work.

What is not implemented. The channel, memory, and supervisor interfaces have no implementation, so the prototype runs a single agent. The journal lives in memory only; there is no persistent backend.

7 Evaluation

We evaluate the Rust prototype (Section 6) on three questions: does replay reproduce a recorded run (E1), does recovery from a crash at any point preserve what was journaled and complete the run (E2), and what does journaling cost (E3)? These experiments test that the implementation follows the record, replay, and resume semantics for a single agent, matching the scope of the formal results, and they use a seeded stand-in for the LLM.

Setup. The agent is a small loop with two branches. It draws `classify` from {`tool` : 5/6, `answer` : 1/6}. On `tool` it draws `tool-arg` from {`arg-A` : 3/5, `arg-B` : 2/5} and performs a durable effect `commit(arg)` under the write-ahead discipline of Section 6; on `answer` it draws `answer`, whose distribution has a single outcome. A recorded run therefore has two or four journal entries. The oracle is `SeededOracle`, a categorical sampler driven by SplitMix64, and the journal is `InMemoryJournal`. All seeds come from a fixed SplitMix64 stream, so every experiment is reproducible. Measurements were taken on an Apple M5 Pro with 48 GB of memory, macOS 27.0.1, and rustc 1.93.0, in release builds. Each experiment is an integration test in the accompanying

repository (`tests/eval_e1.rs`, `eval_e2.rs`, `eval_e3.rs`) run with `cargo test --release --test <name> -- --nocapture`.

7.1 E1: Replay Determinism

For each of $N = 1{,}000$ trials we record a run with one seed, then replay the complete journal with a fresh oracle seeded independently. We compare the traces (each label carries its cursor and the realized value) and the final outputs. Theorem 4.2(i)–(ii) predicts identical traces in every trial.

Result. All 1,000 replays reproduced the recorded trace and output exactly. Every trial made at least one oracle draw, 810 took the branch with the durable effect, and the replay seed differed from the record seed in every trial, so agreement comes from the journal rather than from the oracle. Because replay never consults the live oracle, this outcome is expected by construction; the experiment checks that the implementation conforms to [O-REP], and that a journaled effect is returned from the journal without being performed again.

7.2 E2: Crash Recovery at Every Journal Position

For each of 1,000 recorded runs, and for every crash point k from 0 to the journal length, we keep only the first k journal entries, then run in resume mode with a fresh oracle whose seed differs from the recording seed. If the crash point separates an effect's intent from its committed record, the resumed run stops at the dangling intent; we then apply the first recovery of Section 6, performing the effect again with the journaled argument and committing it, and resume again. For each resume we check that (a) the run completes with an output and leaves no dangling intent; (b) the first k journal entries are unchanged and every trace label whose entry survived is reproduced; (c) every oracle value lies in the support of its distribution and journal cursors stay dense; and (d) an effect whose committed record survived is not performed again.

Result. The sweep made 4,664 resumes. All 4,664 completed and reproduced their surviving prefix, 832 of them crashed inside an effect's write-ahead window and were recovered, and no surviving committed effect was performed again.

Conditional law. Corollary 4.3(ii) says that the resumed continuation follows the original law conditioned on the replayed prefix. We check this on the prefix `classify = tool`: across 20,000 resumes with fresh seeds, the re-sampled `tool-arg` was `arg-A` with frequency 0.5989, against $3/5 = 0.6$ (standard error ≈ 0.0035).

Compensation. After a crash inside the write-ahead window followed by a compensating recovery, replaying the journal twice stops both times at the compensated effect, produces no output, performs no effect, and appends nothing, as the write-ahead design intends.

A defect found by E2. An earlier version of the driver stayed in replay mode after [O-RESUME] fell through to recording, so the next step re-read the entry just appended as if it were history and could produce out-of-support values. The original test suite missed this because its only crash test took the branch whose distribution has a single outcome; the sweep above fails on that version and passes on the corrected one.

Table 2: Summary of the evaluation.

ID	Question	Result relates to	Outcome
E1	Replay determinism	Theorem 4.2	1,000/1,000 identical replays
E2	Crash at every position	Corollary 4.3; side effects	4,664/4,664 resumes correct; no duplicate effects; conditional law matches
E3	Journaling overhead	implementation	87–145 ns per append, flat in n

7.3 E3: Journaling Overhead

Append latency. We append $n \in \{10^3, 10^4, 10^5, 10^6\}$ oracle-draw events to an in-memory journal, after one warm-up pass, and report the mean time per append. Over seven runs, every measurement fell between 87 and 145 ns per append, with no growth in n, consistent with amortized constant-time append; the journal holds exactly one entry per step. These are single timed passes per size, not a statistical benchmark.

Replay versus re-execution. To model a networked LLM call we inject a latency of 1 ms into every oracle draw. A recorded run then takes about 2.51 ms, while replaying it takes 0.5–0.9 μs, a speedup of about 2,700× to 4,900× across seven runs. The speedup is dominated by the injected latency: replay performs no oracle calls, so the time saved per replayed draw is the latency of the call itself.

Not measured. We did not measure the cost of journaling relative to a journal-free execution mode, which the prototype lacks, or the cost of persisting the journal (for example, `fsync` on every append), since no persistent backend exists yet.

7.4 Threats to Validity

Internal. The seeded oracle stands in for an LLM. The experiments show that the implementation follows the record/replay semantics, not how real models behave. Crashes are simulated by truncating an in-memory journal; torn writes, `fsync` behavior, and storage failures are not exercised.

Construct. Trace comparison in E1 and E2 uses labels that carry cursors, prompts, and responses, as in Theorem 4.2. E2 checks the conditional law on one prefix and one prompt, not in general.

External. All measurements come from one machine and a single in-memory backend, and the workload is a small agent; results for persistent backends, larger agents, or many concurrent agents may differ.

8 Related Work

8.1 Formal Models of LLM Agents

LLMbda and FIDES. LLMbda [12] is an untyped call-by-value λ-calculus whose constructs include prompt-response conversations that can be forked and cleared, code generation, and dynamic

information-flow control in which every value carries a label. Its first version followed FIDES [9] in assuming that the underlying model is deterministic; the current version gives a probabilistic semantics, proves a termination-insensitive probabilistic noninterference theorem for the whole calculus, and uses its Lean-verified interpreter as the agent harness that calls the model. FIDES is a planner-based system that enforces information-flow policies around an LLM; its formal guarantees are stated for a deterministic model function. Like LLMbda we treat model calls probabilistically, but our main concern is different: LLMbda studies what an agent's program may do with untrusted data, whereas we study recording, replaying, and recovering a run. Our noninterference theorem (Theorem 5.3) addresses what LLMbda does not model, concurrently scheduled agents, at the price of coarser, per-agent labels.

Other calculi and languages for agents. λ_A [21] extends the simply typed λ-calculus with oracle calls, bounded fixpoints, and probabilistic choice, and proves type safety and termination in Coq. PPDL [22] treats LLM-based flows as probabilistic programs whose semantics is a weighted distribution over traces. Zhou, D'Antoni, and Polikarpova [38] have agents generate well-typed Haskell programs whose information flow is controlled with LIO [30], and prove a soundness theorem with the model treated as a deterministic oracle. None of these addresses journaling, replay, or crash recovery.

MSC coordination calculus. Bollig, Függer, and Nowak [4] specify multi-agent workflows as message sequence charts in a domain-specific language that separates message-passing structure from LLM calls, tool calls, and human control points, project global specifications to deadlock-free local programs, and machine-check all results in Lean 4. Their implementation, ZipperGen, runs workflows through a persistent store and can replay the committed history after a restart, although this durability is not part of the formal model. The calculus establishes coordination properties independently of LLM nondeterminism, treating model outputs as arbitrary; we instead treat each output as a draw from a measure, which is what lets us state that replay and recovery preserve distributions.

Other formal and empirical work. Schlapbach [27] gives a process-calculus semantics for agentic tool protocols such as the Model Context Protocol; the second version of that preprint was withdrawn because of a logical flaw in one theorem, and a revised version has since been posted. Zaichyk [36] studies decentralized coordination over scoped views of an ontology-structured shared memory, with probabilistic results about update proposals, and recovers from failures by re-synchronizing from shared state rather than by replay. Cemri et al. [6] build a taxonomy of multi-agent failures from more than 1,600 annotated traces. Khatchadourian [16] proposes a protocol for replaying tool-using financial agents and comparing runs at different levels; it is an empirical harness, not a semantics. He and Yu [14] argue, in a short position paper, that systems built from agents must tolerate divergent execution traces that reach equivalent outcomes.

8.2 Record, Replay, and Recovery

Message logging. Logging every nondeterministic event so that a process can be replayed after a failure is a classical technique. Strom and Yemini's optimistic recovery [31] and the message-logging protocols surveyed by Elnozahy et al. [11] assume that processes are piecewise deterministic and log the nondeterministic events between deterministic intervals. Consistent global states are due to Chandy and Lamport [7], and compensating transactions to Garcia-Molina and Salem's sagas [13].

Our journal is message logging specialized to an agent whose nondeterministic events are model calls; what we add is a statement of its soundness inside a probabilistic semantics.

Record, replay, and resume for LLM agents. Several recent systems record LLM agent runs at their nondeterministic boundaries and replay them, for debugging or regression testing [8, 23]. Khan [15] specifies a checkpoint-and-resume contract for workflow persistence layers, checks it with machine-checked models, and finds duplicate effects after crashes in widely used frameworks; Zheng et al. [37] give an exact, Lean-mechanized check of when agent executions can be checkpointed, forked, restored, or merged; and Wu et al. [35] show that rollback can restore states that are not consistent causal cuts. Li [20] proves that no policy based on verification alone achieves exactly-once effects under late commits without a bound on in-flight time. These works are deterministic or empirical; ours places replay and recovery inside a probabilistic semantics, for single agents and for systems under deterministic scheduling.

Durable-execution engines. Temporal [32] re-executes deterministic workflow code against an event history and records the results of Activities, including LLM calls, and of replay-safe operations such as random number generation [33]. Restate [25], DBOS [10], and LangGraph's checkpointing [19] take related journaling or checkpointing approaches. To our knowledge none of these systems comes with a formal semantics in which replay soundness is a theorem about distributions.

8.3 Probabilistic Semantics and Guarantees

Our semantics builds on the sampling view of probabilistic programs [5, 17] and uses a coupling argument [2]. For LLM agents, Bensalem et al. [3] argue for chaining probabilistic assume-guarantee contracts to obtain system-level bounds, and Solko-Breslin et al. [29] bound the probability of policy violations without independence assumptions; both are complementary to replay, which preserves whatever probabilities such analyses attach to a run.

9 Limitations and Future Work

The results of this paper assume no recursion, deterministic scheduling with a single journal, and no formalized model of the external world. This section describes the limits of the prototype and what extending the theory requires. Several of these extensions are subtle: designs that look natural turn out to be unsound, and we note the pitfalls we know of.

The prototype. The oracle in our experiments is a seeded stand-in for an LLM, and the journal is in memory, so the experiments show that the implementation follows the semantics, not how it behaves with real models or persistent storage. Experiments with real LLMs and a journal that survives process crashes are the natural next step.

Recursion. Our terms have no recursion, so every run makes finitely many calls. Agents that loop until a goal is met need recursion or a bounded loop; the theorems should extend to finite prefixes of possibly infinite runs, but we have not proved this.

Side effects. Write-ahead logging of side effects is implemented and tested, but only its replay is covered formally (Section 5.3); recovery is not. A formal treatment must state what recovery can guarantee. Performing an interrupted effect again is safe for an idempotent effect of a single agent, but not in general: if another agent changed the same external state in the meantime, performing the effect again can undo that change. Compensation needs its own write-ahead discipline, or a crash during compensation can leave an effect undone twice or not at all. Li [20] shows that exactly-once effects cannot be achieved by verification alone under late commits without a bound on in-flight time, so any recovery theorem must rest on explicit assumptions about the external system, such as idempotence, atomicity, or a reliable way to query whether an effect happened.

Distributed journals and partial restart. Theorem 5.2 assumes deterministic scheduling and one totally ordered journal, and it replays the whole system. Lifting these restrictions is substantially harder. With one journal per agent, persisted independently, recovery must resume from a consistent cut of the journals [7]; with shared memory the cut must account for which write each read observed, not only for messages, and Wu et al. [35] show that agent frameworks can roll back to states that are not consistent cuts. Messages sent before a cut but received after it must then be delivered again exactly once, which requires identifying messages by channel and sequence number rather than by value. Restarting one agent in a running system, as a supervisor might, additionally requires restoring its local state and suppressing the messages it already sent. The prototype runs a single agent, so the multi-agent results are not yet tested.

Information flow. Theorem 5.3 is proved on paper; mechanizing it is the natural next step. Its discipline is coarse: labels are per agent rather than per value, there is no declassification, and side effects, clock reads, and supervisors are excluded. LLMbda [12] tracks labels on individual values and proves probabilistic noninterference for a single agent; APPA [18] adds recoverable floating labels for agents; and Peters-Gill et al. [24] show empirically that single-agent defences against prompt injection do not compose across agents, which is the gap a theorem for concurrently scheduled agents addresses. Adams [1] instead obtains noninterference for language models by running one instance per security level. Combining value-level labels with concurrent scheduling, and relaxing (A-SCHED) to realistic schedulers without reopening timing channels, are open problems.

Compositional guarantees. If each agent of a system meets its own goal with probability at least p_i, Boole's inequality bounds the probability that all do by $1 - \sum_i (1 - p_i)$, without independence. The hard part is establishing the per-agent bounds inside a system, where an agent's inputs depend on other agents and on the scheduler; this needs an assume-guarantee argument of the kind Bensalem et al. [3] advocate, and complements verification approaches that bound the probability of violations directly [29].

10 Conclusion

We gave a minimal probabilistic semantics for durable replay of an LLM agent, in which every model call is a draw from an abstract measure and every response is journaled. In this setting, replaying a run from its journal reproduces it exactly, so record and replay induce the same distribution over traces, and resuming after a crash continues the run with the distribution it would have had without the crash, provided the crash does not depend on a response that was never journaled. The pointwise core of both results is mechanized in Lean 4, and a Rust prototype that implements the semantics passes replay and crash-recovery tests at every journal position. The results place

a classical recovery technique, logging nondeterministic events, inside a probabilistic semantics, so durability no longer requires treating each model call as an opaque value with no distribution. Both results extend to systems of concurrent agents under deterministic scheduling with a single global journal, and such systems satisfy a probabilistic noninterference theorem under a floating-label discipline with fixed labels on memory and channels and a priority scheduler whose priorities depend only on public events. Distributed journals, formalized side effects, finer-grained information flow, and mechanizing the noninterference proof are the main directions for future work.

Acknowledgments and Use of AI Tools

I developed this work with Anthropic's Claude as a collaborator on the proofs, code, and writing. Any errors are my own.

References

[1] Isaiah Adams. Noninterference for language models through secure multi-execution, 2026. Preprint, Zenodo. doi: 10.5281/zenodo.22097472.

[2] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. Relational reasoning via probabilistic coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20)*, volume 9450 of *Lecture Notes in Computer Science*, pages 387–401. Springer, 2015. doi: 10.1007/978-3-662-48899-7_27.

[3] S. Bensalem, Y. Dong, M. Fränzle, X. Huang, J. Kröger, D. Nickovic, A. Nouri, R. Roy, and C. Wu. Position: A three-layer probabilistic assume-guarantee architecture is structurally required for safe LLM agent deployment, 2026. arXiv:2605.18672.

[4] Benedikt Bollig, Matthias Függer, and Thomas Nowak. Provable coordination for LLM agents via message sequence charts, 2026. arXiv:2604.17612v3; accepted at ISoLA 2026.

[5] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 33–46. ACM, 2016. doi: 10.1145/2951913.2951942.

[6] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail?, 2025. arXiv:2503.13657.

[7] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985. doi: 10.1145/214451.214456.

[8] Tisha Chawla and Susheem Koul. Chronicle: Cut-point replay for regression testing of LLM agents, 2026. arXiv:2609.20625.

[9] Manuel Costa, Boris Köpf, Aashish Kolluri, Andrew Paverd, Mark Russinovich, Ahmed Salem, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. Securing AI agents with information-flow control, 2025. arXiv:2505.23643.

[10] DBOS, Inc. DBOS Transact. `https://www.dbos.dev`, 2026. Accessed November 2026.

[11] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002. doi: 10.1145/568522.568525.

[12] Zac Garby, Andrew D. Gordon, and David Sands. The LLMbda calculus: AI agents, conversations, and information flow, 2026. arXiv:2602.20064v2.

[13] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259. ACM, 1987. doi: 10.1145/38713.38742.

[14] Jun He and Deying Yu. Post-deterministic distributed systems: A new foundation for trustworthy autonomous infrastructure, 2026. arXiv:2606.01722.

[15] Sajjad Khan. Resume means resume: A machine-checked conformance contract for checkpoint, interrupt, and resume semantics in workflow persistence layers, 2026. arXiv:2608.03836.

[16] Raffi Khatchadourian. Replayable financial agents: A determinism-faithfulness assurance harness for tool-using LLM agents, 2026. arXiv:2601.15322v4.

[17] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981. doi: 10.1016/0022-0000(81)90036-2.

[18] Arseny Kravchenko, Vadim Liventsev, Innokentii Konstantinov, Ildar Iskhakov, and Matvey Kukuy. APPA: Recoverable information-flow control for real-world LLM agents, 2026. arXiv:2607.24625.

[19] LangChain, Inc. LangGraph: Persistence. `https://docs.langchain.com/oss/python/langgraph/persistence`, 2026. Accessed November 2026.

[20] Jiapeng Li. Where does exactly-once live? Model, harness, and tool-contract effects on duplicate side effects in LLM agents, 2026. arXiv:2609.29095.

[21] Qin Liu. λ_A: A typed lambda calculus for LLM agent composition, 2026. arXiv:2604.11767.

[22] Louis Mandel, Guillaume Baudart, Mandana Vaziri, and Martin Hirzel. PPDL: LLM-based flows as probabilistic programs, 2026. arXiv:2608.05234; published at ICML 2026.

[23] Rasheed Mudasiru. Deterministic replay for AI agent systems, 2026. arXiv:2607.16200.

[24] James Peters-Gill, Avi Semler, Henning Bartsch, Ilia Shumailov, and Christian Schroeder de Witt. Can CaMeLs talk? Securing multi-agent systems against indirect prompt injection attacks, 2026. arXiv:2610.05640.

[25] Restate. Restate. `https://restate.dev`, 2026. Accessed November 2026.

[26] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 200–214. IEEE Computer Society, 2000. doi: 10.1109/CSFW.2000.856937.

[27] Andreas Schlapbach. Formal semantics for agentic tool protocols: A process calculus approach, 2026. arXiv:2603.24747v3.

[28] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. In *CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 1994. doi: 10.1007/978-3-540-48654-1_35.

[29] Alaia Solko-Breslin, Pramod Kaushik Mudrakarta, Mihai Christodorescu, Somesh Jha, and Krishnamurthy Dj Dvijotham. Efficient and sound probabilistic verification for AI agents, 2026. arXiv:2606.20510.

[30] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, pages 95–106. ACM, 2011. doi: 10.1145/2034675.2034688.

[31] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985. doi: 10.1145/3959.3962.

[32] Temporal Technologies, Inc. Temporal workflow definition. `https://docs.temporal.io/workflow-definition`, 2026. Accessed November 2026.

[33] Temporal Technologies, Inc. Side effects (Go SDK). `https://docs.temporal.io/develop/go/workflows/side-effects`, 2026. Accessed November 2026.

[34] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2–3):231–253, 1999. doi: 10.3233/JCS-1999-72-305.

[35] Guanlong Wu, Dahui Li, Ke Jiang, Jianyu Niu, Cong Wang, and Yinqian Zhang. Safe to resume? Breaking execution continuity of agent execution via rollback, 2026. arXiv:2608.29381.

[36] Sofiya Zaichyk. Semantic fusion: Verifiable alignment in decentralized multi-agent systems. *ACM Transactions on Autonomous and Adaptive Systems*, 21(3):1–29, 2026. doi: 10.1145/3797042.

[37] Yusheng Zheng, Xiaoyu Song, Yanpeng Hu, Lebin Cheng, Yuxi Huang, and Wei Zhang. When can agents safely checkpoint, fork, restore, and merge? Exact checking for execution edits, 2026. arXiv:2608.22928.

[38] Timothy Zhou, Loris D'Antoni, and Nadia Polikarpova. Language-based agent control, 2026. arXiv:2605.12863.