Linden: Linear-Typed Agent Collaboration

Anonymous authors
Paper under double-blind review

Abstract

Collaboration among language-model agents is becoming increasingly common, making safe coordination over shared resources a central systems concern. Sandboxing provides a meaningful initial authority boundary, but not a dynamic mechanism for reallocating and transferring authority as collaboration evolves. We present Linden, a system for agent collaboration grounded in linear types. Linden represents resources and their update authority as linear values—values that must be consumed exactly once rather than duplicated—thereby enforcing unique ownership while allowing authority to move between participants. Resource versions and collections follow the same typed ownership discipline. A read-only projection separates shared access from update authority, while a Share monad captures runtime-mediated observations of shared state. Humans, agents, and terminals inhabit a common participant tree whose parent–child relationships constrain resource and authority transfers. We formalize Linden in a core calculus and prove type safety, unique ownership, role authenticity, capability confinement, and atomic observation. An executable language and a runtime authority manager enforce these guarantees during agent execution. Beyond semantic conformance tests and runtime microbenchmarks, we evaluate Linden on general-purpose terminal tasks, small-team coding, and large-scale multi-agent collaboration. On Terminal-Bench 2.1, Linden improves task success by 8.1 percentage points over a conventional sandbox baseline. On CooperBench Flash, Linden-Coop scores 3.4 points above Pi-Coop, while Linden-Team reaches 89.7%, the highest success rate among all evaluated conditions, including Shepherd. On Silo-Bench, Linden reaches 76% success on the common completed set, 28 percentage points above the strongest baseline, and coordinates 100 Agents in a chain-structured task through direct handoffs and independent recomputation.

1 Introduction

Collaboration is becoming an important means of extending the capabilities of language-model agents. OpenAI recently reported a mathematical research effort involving approximately 10,000 concurrent agents in its Navier–Stokes investigation, illustrating the scale at which coordinated agents can operate (OpenAI, 2026). Apodex 1.1 develops coordination as a learned capability, training agents to decompose tasks, delegate parallel work, integrate asynchronous results, and replan (An et al., 2026). Yet scale alone does not produce reliable collaboration: Silo-Bench finds that collective success declines sharply as teams grow, even when the intended communication structure is simple (Zhang et al., 2026). As models increasingly decide how work is organized, the collaboration itself becomes a program constructed during execution.

Current coding-agent systems commonly allocate authority by placing an Agent in a filesystem and network sandbox, with individual tool or command requests subject to additional approval (Microsoft, 2026; Anthropic, 2025). This provides a meaningful initial allocation, but the allocation is largely static: changing sandbox boundaries as work moves between Agents is cumbersome. Approvals can authorize individual exceptions, yet repeatedly approving related operations neither updates the underlying allocation nor records a durable transfer of authority. Because sandbox configuration and approval decisions are maintained

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

as separate mechanisms, they do not fully coordinate how authority should move as Agents spawn, delegate work, and complete their assignments.

Such collaborations need explicit rules for resource ownership, participant lifecycles, and authority transfer. Linear types make these constraints mechanically checkable while leaving models free to construct and revise the collaboration. Prior results show that compiler feedback improves generated programs across model–task configurations and that LLMs can repair real-world Rust ownership and typing errors (Mündler-Sasahara et al., 2026; Deligiannis et al., 2025). This suggests a useful division of labor: models propose evolving programs, while mechanical checks enforce resource constraints.

We introduce Linden[1], a language and runtime for dynamically generated agent collaboration. Each resource is represented by a linear complete value containing its identity, state, and unique update authority; it may be transferred but not duplicated. Copyable pointers permit observation and communication without recovering that authority. Humans, agents, and terminals form a participant tree that governs lifecycle operations and authority transfer, while role pointers support authenticated communication. Authority therefore follows checked structural relationships, whereas messages may follow the topology required by the task.

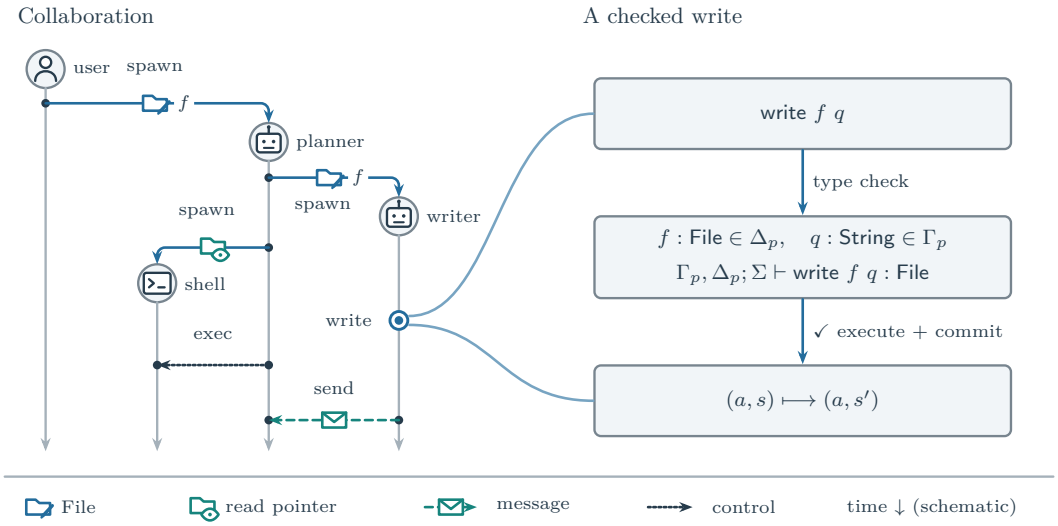


Figure 1: Linden distinguishes ownership, observation, communication, and control within a collaboration.

Figure 1 illustrates the distinction: file ownership moves to the writer, read access is shared with the shell, and execution and communication remain separately controlled.

Our contributions are:

- A core calculus that unifies resource state, linear update authority, shared observation, participant lifecycles, and resource transfer, with proofs of type safety, unique complete ownership, and role authenticity.
- A collaboration language and runtime that implement these rules through checked operations, persistent authority management, and mediated execution.

We evaluate Linden from everyday tool use to large-scale collaboration. On Terminal-Bench 2.1, Linden with delegation improves success by 8.1 percentage points over a conventional

[1]The name combines lin-, for linear types, with the linden tree, evoking both Linden's branching participant structure and the tree's historical role as a place for communal gathering and deliberation (Richter, 2016).

sandbox. On CooperBench Flash, Linden improves both peer and managed-team collaboration, with Linden-Team reaching the highest success rate among all evaluated conditions. On the common completed Silo-Bench set, Linden reaches 76% full-task success, 28 points above the strongest baseline, and completes a 100-Agent dependency chain through direct handoffs and independent recomputation.

2 Related Work

Reliable agent collaboration intersects three lines of work: hierarchical coordination, explicit and reversible execution state, and typed authority. We review these areas in turn and situate Linden by how it combines participant structure with transferable resource ownership.

Hierarchical human and agent collaboration. Hierarchical software collaboration predates LLM agents. Chief programmer teams assigned differentiated implementation and integration responsibilities (Baker, 1972); Linux development routes patches through recursive maintainer hierarchies and a chain of trust (The Linux Kernel Documentation, n.d.). These systems organize work and integration, but do not define the transfer of unique write authority over a shared resource. LLM frameworks make collaboration programmable through conversation patterns in AutoGen (Wu et al., 2023), role-specialized procedures in MetaGPT (Hong et al., 2023), and orchestration and replanning in Magentic-One (Fourney et al., 2024). CooperBench nevertheless identifies communication, commitment, and expectation failures in collaborative coding (Khatua et al., 2026). Linden gives the collaboration structure an explicit authority semantics: Human, Agent, and terminal (Term) participants share one tree; ownership transfers follow direct parent–child edges, and unforgeable role pointers authorize specific participant operations. Humans therefore participate in the same structural discipline rather than appearing only as external approval callbacks. The tree constrains authority rather than message routes: messages carry strings and do not transfer capabilities.

Resource state and reversible execution. Several systems make agent execution state explicit. AgentSPEX specifies typed workflow steps, control flow, explicit state, checkpointing, and logging (Wang et al., 2026). AgentGit adds commit, revert, and branching to multi-agent workflow state (Li et al., 2025). At the operating-system level, BranchFS provides copy-on-write branch contexts with commit or abort and process-group isolation (Wang & Zheng, 2026). Most closely, Shepherd makes tasks, effects, scopes, and execution traces first-class, and supports Git-like reversible agent execution with a mechanized core (Yu et al., 2026). Linden organizes these concerns around a uniform resource representation: a complete value carries stable identity, state, and unique update authority. Its state may be ordinary or a persistent tree with distinguished root and current nodes; reversible resources specialize this representation without changing the ownership discipline. Versioned Files and rewind-capable Agents instantiate the same state-tree structure, which is distinct from the participant tree. Navigation changes only one resource's current node: neither checkout nor rewind! retracts messages, reverses ownership transfers, changes other resources, or reverses IO. An old checkpoint consequently cannot resurrect complete ownership.

Types, ownership, and capability enforcement. Linden builds on linear logic's resource interpretation (Girard, 1987), linear typing for safe destructive update (Wadler, 1990), and the coexistence of linear and unrestricted values (Bernardy et al., 2018). Capability systems ground authority in protected references rather than textual names (Dennis & Horn, 1966). For agent programs, ETAS separates escaping effects from persistent requested-action traces, checks allow, deny, and temporal policies statically, and retains residual runtime checks; effects alone are not permissions (Tan et al., 2026). Agent libOS supports persistent agent processes with capability-controlled operations, attenuating delegation, and durable recovery (Zhang, 2026). Linden focuses on the ownership semantics of collaboration: complete resource values and path-described bundles carry linear update authority, and transferring them consumes the sender's corresponding ownership. Copyable pointers preserve shared observation without duplicating that authority. A common participant tree constrains transfers and lifecycle control across Humans, Agents, and Terms. Linden formalizes these mech-

anisms in a core calculus establishing unique complete ownership, role authenticity, and resource-local revision.

3 Core Theory

Linden represents resource authority directly through linear values rather than inferring it from an operation's effects. Each resource identity admits at most one complete writer, preventing concurrent-write races on that resource while allowing parallel updates to disjoint regions. Reversible state is retained in persistent trees; irreversible effects remain recorded in Agent history. This resource and version model is independent of Git: file access and network IO follow a common authority discipline, and participants can share one workspace under explicit resource grants without requiring separate branches or worktrees.

3.1 Linear Type for Resource and Authority

Linden's types form one hierarchy with two compatible refinements. Resource types are linear types with stable identity and a read-only projection; path resources add set-valued authority, while reversible resources add persistent state trees. Human, Agent, and Term are resource types, while File is both path-addressed and reversible.

Linear values. The typing judgment

$$\Gamma; \Delta \vdash e : C \tag{1}$$

separates an unrestricted context Γ from a linear context Δ. Strings, ordinary pointers, and participant role pointers reside in Γ, where weakening and contraction are admissible. Complete object values and computations whose results contain complete ownership reside in Δ and must be consumed exactly once. A complete value

$$(a, s) : A \tag{2}$$

contains a stable identity a, a state component $s : \mathsf{State}_A$, and the unique authority to update that state. State updates consume one complete value and return one with the same identity; transfer moves a complete value without copying it; spawn! and send create complete values only at fresh participant and message identities. Hence every live identity satisfies $N_\Sigma(a) \leq 1$ (Appendix C). Consequently, a participant that retains complete File authority need not use content hashes to detect another authorized Linden writer.

Resources and read projection. A resource type is a linear type whose complete values have stable identity and admit the non-consuming projection

$$\&(a, s) = a. \tag{3}$$

From $(a, s) : S$, the projection yields $a : \&S$ while retaining the complete value for one later consuming use. Pointers are unrestricted, and there is no rule from $\&S$ back to S. For a fixed identity, the corresponding linear fiber is

$$S_a \triangleq \sum_{s:S} (\&s = a), \qquad S \simeq \sum_{a:\&S} S_a. \tag{4}$$

The pointer does not construct an inhabitant of this fiber. Checkout-capable Files and rewind-capable Agents instantiate the reversible-resource refinement.

Shared observation. Observation through a pointer is runtime-mediated because another participant may publish a new state between observations. Linden records this effect with the Share monad, while preserving linearity whenever the result contains complete ownership. Shared observation may therefore be repeated or concurrent without becoming a source of update authority; the complete rules appear in Appendix A.3.

Reversible resource types. When a resource A retains history, its complete state remains $s : \mathsf{State}_A$, with

$$\mathsf{State}_A \triangleq \mathsf{Tree}^{\bullet\bullet}(\mathsf{Payload}_A), \tag{5}$$

a connected acyclic graph with a distinguished root r, a distinguished current node c, and a $\mathsf{Payload}_A$ label at each node. For Files, the node payload is $\mathsf{Payload}_{\mathsf{File}} = \mathsf{Absent} \mid \mathsf{Present}(\mathsf{String})$. Navigation selects an existing $n \in V$ as current while preserving the graph and labels. Growth adds one fresh child of the current node and makes it current, so growth after navigation creates a branch without discarding old nodes. A versioned File instantiates navigation as checkout and growth as write; an Agent instantiates navigation as rewind!. Rewind changes only the Agent's current state node: messages, transfers, other resources, and IO remain unchanged. Forward or compensating edge functions may realize this transition, but they are not additional values or operations in the calculus.

Path resource types. Path resources lift authority from individual identities to sets generated by path cones, union, intersection, difference, and singleton. Linden normalizes these descriptions into disjoint regions. Before an address carries its own state, it is split from the owning bundle: $S = \{a\} \sqcup (S - \{a\})$. The exact resource then carries the state for a, while the residual bundle explicitly excludes it. This refinement partitions existing authority without creating privilege; Appendix A.2 gives the canonical form. The same construction represents Web authority over host selectors.

3.2 Participant Tree and Structural Authority

Resource types determine what may be owned and observed; participant roles determine who may transfer that authority and control its lifecycle. Human, Agent, and Term are resource types arranged in one parent–child tree. Agent supplies conversational and reversible behavior, Term supplies sandboxed command execution, and Human participates through the same authority model. For participant types P, Q, structural role pointers distinguish a direct child $\&_{>}P$, a descendant $\&_{>>}P$, and a direct parent $\&_{<}Q$. These roles arise from the live tree recorded in runtime state and determine which participant operations are available. They are capabilities justified by that relation, not the source from which the relation is reconstructed; discarding a role does not remove its edge. An ordinary pointer or displayed identifier cannot construct them.

spawn! creates a fresh child and the roles justified by its new edge, atomically moving selected complete resources from the parent to the child. Ownership subsequently moves only across direct edges: give! transfers a complete value in either direction, while parent-side get! retrieves one held by a direct child. A child cannot take a resource from its parent, and descendant roles do not authorize transfers. Provisioning and transfer preserve the resource's unique complete ownership. A parent can also rewind! a direct Agent child and archive! a stopped direct child, authorizing the local state navigation described above.

The same participant structure also unifies ordinary Agent returns and authority coordination. An Agent reports results by sending a Message to its parent and may use the same channel to request additional authority. The request itself grants no authority; any subsequent transfer remains governed by the participant tree.

Appendices A–C give the complete rules and establish preservation, relative progress, unique complete ownership, capability confinement for send, role authenticity, version-tree preservation, checkout locality, and atomic observation.

The next section follows these judgments through the programming interface and runtime, using a collaboration task to connect resource authority to execution.

4 Programming and Execution

Linden exposes the operations of Section 3 through an imperative language and participant-facing tools. The runtime checks and executes their context transitions across files, model sessions, and terminal processes. Figure 2 connects this execution flow to partitioned write delegation and recursive read-only analysis.

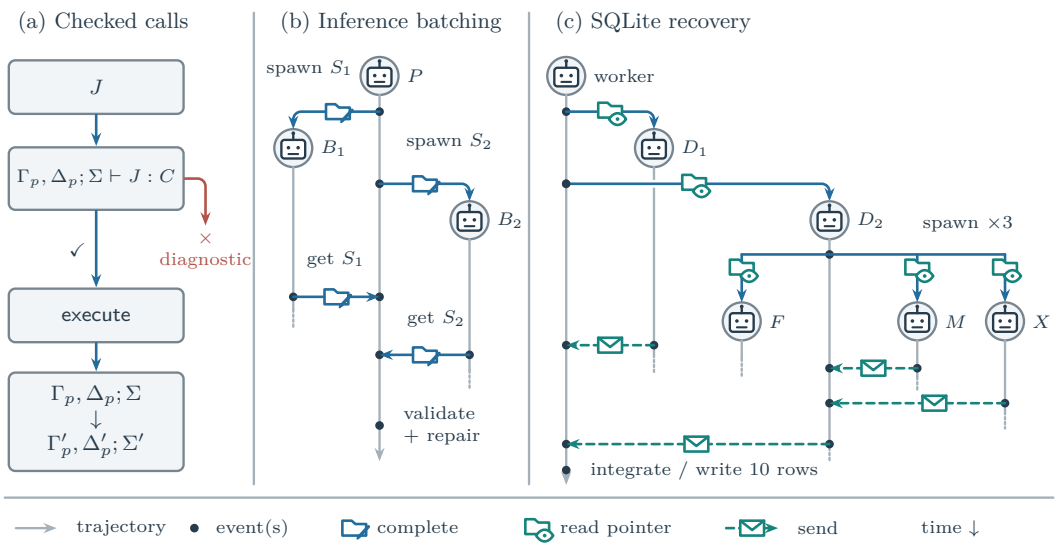


Figure 2: Checked execution and two collaboration patterns. (a) Calls execute and commit only after checking. (b) A parent delegates disjoint staging regions and reclaims them for integration. (c) Read-only analysts report to a single output owner.

4.1 Programming Language

Commands lower through Source → AST → Checked IR. For example, write a q elaborates to a = write a q, consuming the complete File and rebinding its successor. Projection &a retains the complete binding. Participant-facing tools resolve arguments to the same typed terms, allowing calls to be checked incrementally during a collaboration. A reference such as file:... names a resource whose usable authority comes from the caller's context. Auto-refinement partitions a held region into selected resources and residual authority; transferring the selection leaves the sender with the residual.

The checker tests ownership, structural roles, lifecycle premises, and path refinement against the current contexts and a read-only view of published state. A successful check produces a result type, proposed post-contexts, and an effect classification; it does not update the live contexts. Appendix D specifies the surface forms, typed references, and checking rules.

4.2 Runtime Enforcement

One session authority manager implements the sequence in Figure 2(a). It first checks each call against the caller's current authority and the published state. If the check succeeds, it executes the concrete adapter operation; only if execution succeeds does it commit the proposed post-contexts together with the published state. Rejected calls do not execute, and a failed operation does not commit the proposed post-contexts. A failed native File operation preserves the prior binding. Human commands enter the same manager under the Human participant's authority. Path canonicalization, sandbox configuration, and filesystem reconciliation enforce these decisions on physical resources.

External commands run through Term participants with authority fixed for an execution epoch. The bash tool provides a temporary Term lifecycle. The sandbox restricts resource access during execution, while concurrent native File reads return the last committed snapshot. Once foreground and background work is quiescent, reconciliation publishes supported File changes as one atomic multi-resource transition. Conflicting authority transfers wait for this boundary. Agent checkpoints and File versions support the local revision operations of Section 3.1. Appendix E gives the epoch and history protocols.

Trace analyses of the batching and SQLite-recovery collaborations in Figure 2 appear in Appendices G.1 and G.2.

5 Evaluation

We study enforcement (RQ1), runtime overhead (RQ2), task utility and dynamic delegation (RQ3), small-team coding collaboration (RQ4), and large-scale multi-agent collaboration (RQ5). Appendix F gives the experimental procedures and detailed results.

5.1 Safety and Runtime Overhead

RQ1: Semantic enforcement. Linden passes all 117 checker and runtime tests, including 28 tests targeting ownership transfer, role authenticity, path isolation, revision, failure atomicity, persistence, and physical enforcement. The suite confirms that broader path grants cannot bypass transferred File authority, rewind does not reverse completed transfers, rejected operations are atomic, and network access requires Web authority. Appendix F.1 gives the complete breakdown.

RQ2: Runtime overhead. We benchmark 40 cases across nine operation families on one host with model calls disabled. Table 1 reports representative native and Linden latencies; Appendix F.2 gives the sampling procedure and complete results.

Table 1: Runtime latency and absolute overhead for common operations. Added latency is the difference between Linden and native p50.

Operation	Unit	Native p50	Linden p50	Linden p95	Added latency
Read	μs	2.14	7.91	11.87	+5.77
Write	ms	1.68	5.64	6.56	+3.96
Edit	ms	1.55	5.78	6.85	+4.23
Bash, new sandbox	ms	5.57	24.83	26.21	+19.26

Linden adds 5.77 μs per read and approximately 4 ms per versioned write or edit, and 19.26 ms when creating a Bash sandbox.

5.2 Task Utility and Dynamic Delegation

RQ3: Task completion. We evaluate Terminal-Bench 2.1 (Merrill et al., 2026; The Terminal-Bench Team, 2026) with two repetitions per task under four conditions: Pi-Naive, Sandbox, Linden without additional Agent creation, and Linden with delegation. Conditions share the model, prompt, and task-specific deadline. Figure 3 (left) reports the 74 task–repetition pairs graded in every condition. Linden + Delegation reaches 67.6% success, 8.1 points above Sandbox and within 1.4 points of Pi-Naive. Failed tool calls, including authority-boundary rejections, are analyzed in Appendix F.3. Trace inspection confirms that authority enforcement introduces friction: Sandbox records 42 static or OS-level permission failures, while Linden + Delegation records 87 role-topology rejections and 19 unauthorized file-access attempts. These rejections make the enforced safety boundaries observable, but also add coordination and recovery overhead relative to the less constrained Pi-Naive condition.

Natural delegation. Among 70 Linden + Delegation trials with runtime metrics in the paired set, 67.1% create additional Agents. In a successful MIPS-interpreter trial, the parent delegates executable analysis, source analysis, and runtime research to three read-only children. Their reports identify a custom syscall convention, a 1 GB BSS that calls for sparse memory, and MIPS32r2 delay-slot semantics; the parent integrates these constraints into an implementation that passes all grader checks. Appendix F.3 gives the creation-count and depth distributions.

File recovery. Versioned files support recovery after destructive tool effects. In both WAL-recovery repetitions, the WAL file becomes absent after a SQLite probe. The Agent inspects the File's version history, checks out its initial contents, decodes the WAL, and produces the recovered records. Both repetitions pass all seven grader checks. Capturing versions around Bash execution preserves the input needed to continue after the destructive probe.

Persistent execution. Terms support another recurring requirement: keeping processes alive across Agent interactions. They are created in 33.8% of the 74 paired Linden trials. All four Linden model-inference trials use Terms with a single Agent and pass all four service checks, including API availability. Together, these results show how delegation, versioned files, and persistent processes support different parts of task execution. Across 70 usage-complete pairs, aggregate token consumption under Linden + Delegation is a median 2.32 times that under Linden.

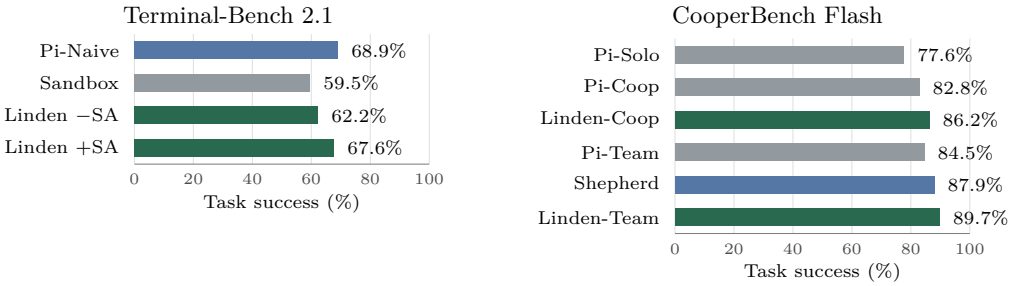


Figure 3: Task success in two evaluations. Left: 74 commonly graded Terminal-Bench task–repetition pairs. Right: the common valid CooperBench Flash set.

5.3 Small-Team Coding Collaboration

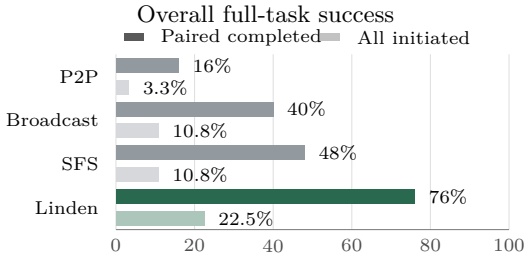
RQ4: Small-team collaboration. On CooperBench Flash, we compare Pi-Solo, peer and lead–member Pi teams, corresponding Linden teams, and Shepherd (Khatua et al., 2026; Yu et al., 2026). All conditions use the same model, reasoning level, task inputs, and feature graders; Figure 3 (right) reports the common valid set, and Appendix F.4 gives the complete protocol. Linden-Team reaches 89.7%, exceeding Pi-Team by 5.2 points and Shepherd by 1.7; Linden-Coop exceeds Pi-Coop by 3.4 points.

Integrating overlapping changes. CooperBench requires independently assigned feature work to survive in one joint patch, especially when features touch the same source file. In the Jinja loader task, path normalization and automatic cache invalidation both modify src/jinja2/loaders.py. In both repetitions, Pi-Team's lead treats the member's cache-invalidation work as outside the requested scope and removes it, so the resulting patches pass path normalization but fail cache invalidation. Linden-Team retains both assignments and transfers the file first to the normalization worker and then to the cache-invalidation worker. Both resulting patches preserve the two implementations and pass both feature graders, showing how explicit ownership handoffs turn overlapping edits into an ordered integration sequence.

5.4 Large-Scale Multi-Agent Collaboration

RQ5: Collaboration at scale. We compare Linden with Silo-Bench's P2P, Broadcast, and shared-file-system conditions at $N \in \{2, 10, 50, 100\}$ Agents (Zhang et al., 2026). Conditions share models, instances, budgets, and deadlines, and each Agent begins with one private shard. Figure 4 reports both the common completed set and all initiated trials; Appendix F.5 gives complete execution status and scale breakdowns.

Linden reaches 76% success on the common completed set, 28 points above SFS, and solves every instance solved by any baseline. Across all initiated trials, it reaches 22.5%, versus 10.8% for the strongest baselines. The gain is concentrated in aggregation and local-



Full-task success by task family

	P2P	Bcast	SFS	Linden
Aggregation	2/40	8/40	8/40	14/40
Local dependencies	1/40	2/40	3/40	9/40
Global reorganization	1/40	3/40	2/40	4/40

All initiated; 40 trials per cell.

Figure 4: Full-task success on Silo-Bench. Left: common completed instances and all initiated trials. Right: all-initiated results by task family; success requires every Agent to be correct.

dependency tasks, while global reorganization remains close. Linden uses more recorded tokens but reduces median wall-clock time by 66–69% relative to the round-synchronous baselines.

Flexible communication under checked authority. Agent pointers identify message recipients, while structural roles expose participant relationships and govern lifecycle and authority transfer. Read-only pointers weaken authority-bearing values into copyable observation capabilities. Linden can therefore construct task-specific communication overlays while keeping update authority unique and structurally checked.

The task families exercise two consequences of this design. Aggregation tasks require a local fact to reach every decision maker: in Any Match at $N = 10$, Agent 8 sends the only positive observation directly to its nine peers, and Linden reaches 10/10 correct Agents, compared with 8/10 for SFS, 6/10 for P2P, and 4/10 for Broadcast. Local-dependency tasks instead require task-shaped handoffs: in Pipeline Hash, Agent i exchanges chain state with adjacent peers, while delayed predecessors can be handled by independently recomputing the public prefix. The same overlay scales to the 100-Agent run, where every Agent produces the correct output after 346 successful direct sends. In both cases, the message graph follows the computation without transferring ownership of the private shards.

6 Conclusion

Linden represents collaboration authority as typed program state: linear values preserve unique update authority, read-only pointers permit shared observation, and participant roles govern lifecycle operations and transfer. Its calculus, language, and runtime carry these guarantees from formal semantics to execution.

Linden passes all 117 enforcement tests with approximately 4 ms of added latency for versioned writes and edits. Delegation improves Terminal-Bench success by 8.1 points over Sandbox; Linden-Team leads the evaluated CooperBench conditions; and Linden exceeds the strongest Silo-Bench baseline by 28 points on the common completed set while completing a 100-Agent dependency chain.

Through the Curry–Howard correspondence, proof obligations may be represented as linear resources and dynamically delegated among cooperating prover Agents, extending Linden toward multi-prover systems.

A separate direction is controlled recursive self-improvement: lower-level Agents may optimize delegated components while higher-level participants retain authority over their evaluation and integration.

AI Use Statement

AI tools were used to assist with software development and manuscript drafting. The authors reviewed the resulting code, mathematical statements, citations, experimental records, and prose, and remain responsible for the paper's claims.

Reproducibility Statement

The static semantics, dynamic semantics, and proof sketches appear in Appendices A–C. The language and runtime boundary is documented in Appendices D and E, Appendix F records the experimental conditions, metrics, and artifact requirements, and Appendix G traces the collaboration examples. The implementation and tests are available anonymously at https://anonymous.4open.science/r/linden/README.md. The accompanying supplementary material provides trial-level outcomes and scripts for recomputing the reported aggregate results.

References

B. An, B. Li, B. Wang, B. Zhang, B. L. Wang, C. Feng, C. Wei, C. Xue, C. Zhang, D. Ng, D. Ye, E. Min, F. Chen, F. Liu, F. Yang, F. Ye, G. Sun, H. Ji, H. Xu, H. Yang, H. Ye, H. Zhang, H. Zhao, J. Li, J. Lin, J. Xia, K. Jin, K. Wang, K. Yang, L. Bing, L. Lei, L. Su, Le. Wang, Lu. Wang, N. Wang, Q. Ren, Q. Yang, R. Li, S. Bai, S. Du, S. Li, S. Lin, S. Nie, S. Wang, S. Zhang, S. Z. Wang, T. Ge, Ta. Q. Fang, Ti. Q. Fang, W. Fang, W. Li, W. Zhang, X. Chen, X. Li, X. Tang, X. Wang, X. Xu, X. Zhang, X. Q. Wang, X. Y. Wang, Y. Deng, Y. Gao, Y. Hu, Y. Li, Y. Sui, Y. Wang, Y. Xiao, Y. Zhang, Y. Zhou, Z. Chen, Z. Cheng, Z. Feng, Z. Liang, Z. Liu, and Z. Zhang. Apodex 1.1: Scaling agentic intelligence for complex work, 2026. URL https://arxiv.org/abs/2608.23283.

Anthropic. Making claude code more secure and autonomous with sandboxing. Anthropic Engineering, 2025. URL https://www.anthropic.com/engineering/claude-code-sandboxing. Accessed 2026-09-21.

F. Terry Baker. Chief programmer team management of production programming. IBM Systems Journal, 11(1):56–73, 1972. URL https://www.bitsavers.org/pdf/ibm/IBM_Systems_Journal/111/ibmsj1101E.pdf.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. Proceedings of the ACM on Programming Languages, 2(POPL):1–29, 2018. doi: 10.1145/3158093.

Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, Rishi Poddar, and Aseem Rastogi. RustAssistant: Using LLMs to fix compilation errors in Rust code. In Proceedings of the 47th International Conference on Software Engineering (ICSE), 2025. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2024/08/paper.pdf.

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. Communications of the ACM, 9(3):143–155, 1966. doi: 10.1145/365230.365252.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-one: A generalist multi-agent system for solving complex tasks, 2024. URL https://arxiv.org/abs/2411.04468.

Jean-Yves Girard. Linear logic. Theoretical Computer Science, 50(1):1–101, 1987. doi: 10.1016/0304-3975(87)90045-4.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng

Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2023. URL https://arxiv.org/abs/2308.00352.

Arpandeep Khatua, Hao Zhu, Peter Tran, Arya Prabhudesai, Frederic Sadrieh, Johann K. Lieberwirth, Xinkai Yu, Yicheng Fu, Michael J. Ryan, Jiaxin Pei, and Diyi Yang. Cooperbench: Why coding agents cannot be your teammates yet, 2026. URL https://arxiv.org/abs/2601.13295.

Yang Li, Siqi Ping, Xiyu Chen, Xiaojian Qi, Zigan Wang, Ye Luo, and Xiaowei Zhang. Agentgit: A version control framework for reliable and scalable llm-powered multi-agent systems, 2025. URL https://arxiv.org/abs/2511.00628.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Microsoft. Manage approvals and permissions for agents in visual studio code. Visual Studio Code Documentation, 2026. URL https://code.visualstudio.com/docs/agents/run/approvals. Accessed 2026-09-21.

Niels Mündler-Sasahara, Hristo Venev, Dawn Song, Martin Vechev, and Jingxuan He. Generative compilation: On-the-fly compiler feedback as AI generates code, 2026. URL https://arxiv.org/abs/2607.13921.

OpenAI. On the Navier–Stokes millennium prize problem. OpenAI Research, 2026. URL https://openai.com/index/navier-stokes-solution/. September 8, 2026. Accessed September 12, 2026.

Gerhard Robert Richter. Zur kulturgeschichte der linde. LWF Wissen 78, Bavarian State Institute of Forestry, 2016. URL https://www.lwf.bayern.de/wissenstransfer/forstliche-informationsarbeit/144564/index.php. Accessed 2026-09-21.

Huiri Tan, Yikun Wang, Puyang Zhang, Shangyu Li, and Jiasi Shen. Etas: An effect-typed language for agent systems, 2026. URL https://arxiv.org/abs/2607.17780.

The Linux Kernel Documentation. How the development process works, n.d. URL https://docs.kernel.org/process/2.Process.html. Section 2.3: How patches get into the Kernel. Accessed September 7, 2026.

The Terminal-Bench Team. Terminal-bench 2.1. Terminal-Bench, 2026. URL https://www.tbench.ai/news/terminal-bench-2-1. May 6, 2026. Accessed September 25, 2026.

Philip Wadler. Linear types can change the world! In Programming Concepts and Methods. North-Holland, 1990.

Cong Wang and Yusheng Zheng. Fork, explore, commit: Os primitives for agentic exploration, 2026. URL https://arxiv.org/abs/2602.08199.

Pengcheng Wang, Jerry Huang, Jiarui Yao, Rui Pan, Peizhi Niu, Yaowenqi Liu, Ruida Wang, Renhao Lu, Yuwei Guo, and Tong Zhang. Agentspex: An agent specification and execution language, 2026. URL https://arxiv.org/abs/2604.13346.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

Simon Yu, Derek Chong, Ananjan Nandi, Dilara Soylu, Jiuding Sun, Christopher D. Manning, and Weiyan Shi. Shepherd: Enabling programmable meta-agents via reversible agentic execution traces, 2026. URL https://arxiv.org/abs/2605.10913. Version 3.

Yingqi Zhang. Agent libOS: A runtime substrate for capability-controlled self-evolving LLM agents, 2026. URL https://arxiv.org/abs/2606.03895v3.

Yuzhe Zhang, Feiran Liu, Yi Shan, Xinyi Huang, Xin Yang, Yueqi Zhu, Xuxin Cheng, Cao Liu, Ke Zeng, Terry Jingchen Zhang, and Wenyuan Jiang. Silo-bench: A scalable environment for evaluating distributed coordination in multi-agent llm systems, 2026. URL https://arxiv.org/abs/2603.01045.

A Static Semantics

For each live participant p, let Γ_p and Δ_p be its unrestricted and linear contexts, and write $\Xi \triangleq \{\Gamma_p; \Delta_p\}_{p \in \mathcal{P}}$ for the finite context map. The local judgment is $\Gamma_p; \Delta_p \vdash e : C$. Strings, ordinary pointers, and participant role pointers reside in Γ_p; complete object values and computations whose results contain complete ownership reside in Δ_p. Weakening and contraction are admissible only in the former.

Operations that may change several participant contexts use the abstract input/output judgment

$$\Xi; \Sigma\text{-view} \vdash_p J : C \dashv \Xi'; \mathit{effect}, \tag{6}$$

where p invokes J. The primed contexts and allowed effect are an intermediate checked result, not a new semantic state. Only successful realization commits Ξ' together with the corresponding Σ'. Rejecting or abandoning the proposal before commit leaves the corresponding contexts and published states unchanged. The core calculus abstracts from how an implementation performs, waits for, or retries that realization.

A.1 Types and Linear Contexts

The selected types are

$$\begin{aligned} C, D \quad ::= \quad & \mathbf{1} \mid \mathsf{String} \mid \mathsf{File} \mid \mathsf{Human} \mid \mathsf{Agent} \mid \mathsf{Term} \mid \mathsf{Message} \\ & \mid \mathsf{Bundle}_A(R) \mid \&S \mid \&_{>}P \mid \&_{>>}P \mid \&_{<}P \\ & \mid C \otimes D \mid C \multimap D \mid \mathsf{Share}\, C \mid \mathsf{IO}\, C, \end{aligned} \tag{7}$$

where S ranges over resource types, P over participant types, A over path-addressed resource types, and $\mathsf{Bundle}_A(R)$ is authority over atoms in R. Selected value forms are

$$v \quad ::= \quad () \mid q \mid (a, s) \mid a \mid (a, \rho) \mid \mathsf{bundle}_A(R) \mid (m, \mathsf{Msg}(q)). \tag{8}$$

The typing derivation distinguishes an ordinary identity $a : \&A$ from a complete value $(a, s) : A$, while $\rho \in \{>, >>, <\}$ distinguishes structural roles. Complete values, Messages, and bundles are linear. Strings, unit, ordinary pointers, and role pointers are unrestricted. Products and computations are linear whenever they contain a linear component; in particular, $\mathsf{Share}\, C$ is linear when C is.

The standard linear rules split Δ disjointly between premises and substitute a linear value exactly once. State components contain no update authority. For a reversible resource, $\mathsf{Payload}_A$ is additionally capability-free: it contains neither a complete value nor a participant role. This restriction prevents navigation to an old payload from restoring authority. A path description denotes atoms but is not a second authority.

Identity projection is the distinguished non-consuming observation of a linear entry:

$$\frac{x : S \in \Delta \qquad S \ \mathsf{resource}}{\Gamma; \Delta \vdash \&x : \&S}. \tag{9}$$

The conclusion retains the same Δ. No rule converts $\&S$ back into S.

The structural conversions are one-way:

$$\frac{\Gamma; \Delta \vdash e : \&_{>}P}{\Gamma; \Delta \vdash e : \&_{>>}P}, \qquad \frac{\Gamma; \Delta \vdash e : \&_{>}P}{\Gamma; \Delta \vdash e : \&P}, \qquad \frac{\Gamma; \Delta \vdash e : \&_{<}P}{\Gamma; \Delta \vdash e : \&P}. \tag{10}$$

These rules forget directness or direct-edge authority. There are no converse rules and no erasure from $\&_{>>}P$ to $\&P$.

A.2 Path Resource Descriptions

For identities of a resource A ordered by a subpath relation $\leq$, define

$$\mathsf{Set}(\&A) \triangleq \&A \to \mathsf{Prop}, \qquad \mathsf{path}_A(\pi)(a) \triangleq (a \leq \pi). \tag{11}$$

Cones generated by path expressions form a laminar family. Sets produced by union, intersection, difference, and singleton therefore have the canonical form

$$\bigsqcup_i \big(\mathsf{path}_A(\pi_{i,0}) - \mathsf{path}_A(\pi_{i,1}) - \cdots - \mathsf{path}_A(\pi_{i,k_i})\big). \tag{12}$$

A value $\mathsf{bundle}_A(R) : \mathsf{Bundle}_A(R)$ compactly records complete authority over the atoms in R but stores no per-address state. When a successful operation materializes $a \in R$ with $s : \mathsf{State}_A$, the calculus uses the meta-level refinement

$$\mathsf{bundle}_A(R) \rightsquigarrow_a (a, s) : A \otimes \mathsf{bundle}_A(R - \{a\}). \tag{13}$$

This is bookkeeping within the same successful atomic transition, not a source primitive or an intermediate state. The calculus constrains the result to have type State_A but leaves the concrete production of s to the resource adapter.

For a context map Ξ, define complete-authority coverage by

$$\mathsf{cov}_\Xi(a) \triangleq \#\{(a, s) : A \text{ in } \Xi\} + \#\{\mathsf{bundle}_A(R) \text{ in } \Xi \mid a \in R\}. \tag{14}$$

Exact values and complete bundles must have pairwise-disjoint coverage. Unrestricted pointers are not counted and may overlap. For Files, $\mathsf{Payload}_{\mathsf{File}} = \mathsf{Absent} \mid \mathsf{Present}(\mathsf{String})$, so a stable materialized location may have an absent payload. The two system instances are $\mathsf{path}_{\mathsf{File}}$ for file locations and $\mathsf{path}_{\mathsf{Web}}$ for host selectors ordered by subdomain.

A.3 Share and IO Computations

The effect operations are

$$\begin{aligned} \mathsf{return} &: C \multimap \mathsf{Share}\, C, \\ \mathsf{bind} &: \mathsf{Share}\, C \otimes (C \multimap \mathsf{Share}\, D) \multimap \mathsf{Share}\, D. \end{aligned} \tag{15}$$

If C contains complete ownership, then $\mathsf{Share}\, C$ is itself linear. This prevents duplication of a suspended get! or any other computation that will produce a complete value. Share therefore accounts for effectful observation but is neither an inferred authority summary nor a substitute for ownership.

The sole cross-effect rule is

$$\frac{\Gamma; \Delta \vdash e : \mathsf{Share}\, C}{\Gamma; \Delta \vdash e : \mathsf{IO}\, C}. \tag{16}$$

There is no converse rule, so composition in Share cannot introduce an explicitly external operation.

A.4 Resource, Participant, and Communication Operations

Messages are consumable linear information:

$$\mathsf{Message} ::= \mathsf{Msg}(\mathsf{String}), \qquad (m, \mathsf{Msg}(q)) : \mathsf{Message}. \tag{17}$$

A single receive consumes a finite batch selected by an abstract relation $\mathsf{select}_p(h, \Xi)$, where the optional h restricts the sender, and returns its deterministic String encoding. The core theory uses only finiteness and linear consumption; ordering, delivery classes, and queue representation belong to the runtime refinement.

The principal ordinary signatures are

$$\begin{aligned}
\mathsf{read} &: \&\mathsf{File} \to \mathsf{Share}\,\mathsf{Payload}_{\mathsf{File}},\\
\mathsf{write} &: \mathsf{File} \otimes \mathsf{String} \multimap \mathsf{File},\\
\mathsf{edit} &: \mathsf{File} \otimes \mathsf{String} \otimes \mathsf{String} \multimap \mathsf{File},\\
\mathsf{listen} &: \&_{>>}P \to \mathsf{Share}\,\mathsf{String},\\
\mathsf{wait}_{\rho} &: \&_{>>}P \to \mathsf{Share}\,\mathbf{1},\\
\mathsf{interrupt} &: \&_{>>}P \to \mathsf{Share}\,\mathbf{1},\\
\mathsf{stop} &: \&_{>>}P \to \mathsf{Share}\,\mathbf{1},\\
\mathsf{reason} &: \mathsf{Share}\,\mathsf{String},\\
\mathsf{send} &: \&H \otimes \mathsf{String} \to \mathsf{Share}\,\mathbf{1},\\
\mathsf{send} &: \&_{>>}H \otimes \mathsf{String} \to \mathsf{Share}\,\mathbf{1},\\
\mathsf{receive} &: \to \mathsf{Share}\,\mathsf{String},\\
\mathsf{receive} &: \&H \to \mathsf{Share}\,\mathsf{String},\\
\mathsf{steer} &: \&_{>>}\mathsf{Agent} \otimes \mathsf{String} \to \mathsf{Share}\,\mathbf{1},\\
\mathsf{exec} &: \&_{>>}\mathsf{Term} \otimes \mathsf{String} \to \mathsf{IO}\,\mathbf{1},\\
\mathsf{input} &: \&_{>>}\mathsf{Term} \otimes \mathsf{String} \to \mathsf{IO}\,\mathbf{1},
\end{aligned} \tag{18}$$

where H ranges over Human and Agent, and $\rho \in \{\mathsf{ready}, \mathsf{stopped}\}$. Read observes $\mathsf{Payload}_{\mathsf{File}}$ at the current node and does not consume its pointer. Write and edit consume one complete File and return one with the same identity; edit additionally requires a present state containing the designated occurrence. Lifecycle predicates are abstract availability premises here: they determine whether wait, exec, input, or archive has a rule, but the calculus does not model the scheduler or the intermediate process states by which an implementation establishes them.

Reason is available only in the current Human or Agent context. Send targets a Human or Agent and creates a fresh Message without changing the target's complete value. Receive is available only to Human or Agent; its optional sender filter requires an ordinary Human/Agent pointer. The separate descendant-target send rule does not manufacture such a pointer. Steer is Agent-specific. Exec requires a ready Term and input requires a running Term.

Log is available to an Agent for itself and to any participant for a strict descendant Agent or Term:

$$\frac{[a : \&\mathsf{Agent}]_p \qquad a = p}{\mathsf{log}_p(a) : \mathsf{Share}\,\mathsf{String}} \qquad \frac{[(a, >>) : \&_{>>}\mathsf{Agent}]_p}{\mathsf{log}_p((a, >>)) : \mathsf{Share}\,\mathsf{String}} \qquad \frac{[(t, >>) : \&_{>>}\mathsf{Term}]_p}{\mathsf{log}_p((t, >>)) : \mathsf{Share}\,\mathsf{String}}. \tag{19}$$

Agent log exposes its published version-tree observation and scoped state identifiers. Term log exposes completed epoch and command records. Neither form grants authority, and Human history is not defined by the calculus.

The privileged give! macro transfers a complete resource or grants a read pointer across a direct edge:

$$\begin{aligned}
&\mathsf{give!} : \&_{\sim}P \otimes S \multimap \mathsf{Share}\,\mathbf{1},\\
&\mathsf{give!} : \&_{\sim}P \otimes \&S \to \mathsf{Share}\,\mathbf{1}, \qquad \sim \in \{<, >\}.
\end{aligned} \tag{20}$$

Parent-side get! is indexed by the selected stable identity:

$$\frac{[(a, >) : \&_{>}P]_p \qquad [(b, s) : S]_a \qquad S\ \mathsf{resource}}{\mathsf{get!}_p((a, >), b) \Downarrow (b, s) : S}. \tag{21}$$

The selector b is resolved at the tool/runtime boundary; it is not an additional source capability. A child has no symmetric rule for taking a complete value from its parent.

A.5 Participant Tree Methods

The live parent–child relation is structural runtime state. Role pointers are capabilities derived from and checked against that relation; retaining or discarding a role neither creates nor removes an edge.

Let $p : Q$ spawn a fresh child $a : P$ with initial state s_0. The atomic structural postcondition is

$$[(a, s_0) : P]_a, \qquad [(a, >) : \&_> P]_p, \qquad [(p, <) : \&_< Q]_a, \tag{22}$$

together with

$$[(a, >>) : \&_{>>} P]_h \qquad \text{for every } h \longrightarrow^+ p. \tag{23}$$

The same transition provisions selected permissions. It moves each selected complete value from parent to child and may make existing unrestricted non-role permissions available according to their unrestricted status. It creates no other role pointer. Freshness prevents a second complete participant value for an existing identity.

The common archive! method requires $(a, >) : \&_> P$ and $\mathsf{Stopped}(a)$. It atomically folds the stopped subtree from its leaves. Here $\mathsf{Stopped}$ is an abstract availability premise; its concrete realization is outside the calculus. At each recursive stage, a leaf's complete external resources move across its direct edge to its parent; that leaf's internal participant and Message values are consumed, and all roles justified by the removed edge are removed from the abstract context map. Repeating this adjacent fold moves the remaining external authority across the original direct-child edge to the caller. The recursion is not externally observable: the whole fold commits as one macro transition. Unrestricted non-role external permissions held only in removed contexts are projected to the caller, whereas existing ordinary pointers elsewhere are not revoked. A descendant role and a merely ready child do not satisfy the rule.

A.6 Version Navigation

For a reversible resource A, its complete state is $\mathsf{State}_A \triangleq \mathsf{Tree}^{\bullet\bullet}(\mathsf{Payload}_A)$. If $(a, s) : \mathsf{File}$, $s : \mathsf{State}_{\mathsf{File}}$, and n belongs to the node set V of s, then

$$\mathsf{checkout}((a, s), n) : \mathsf{File}. \tag{24}$$

This rule schema is indexed by a mathematical node of the particular state tree; it introduces no polymorphic node type.

If $(a, >) : \&_> \mathsf{Agent}$ is held by the direct parent, the child's state is $\mathsf{State}_{\mathsf{Agent}} = \mathsf{Tree}^{\bullet\bullet}(\mathsf{Payload}_{\mathsf{Agent}})$, and n belongs to that tree, then

$$\mathsf{rewind!}((a, >), n) : \mathsf{Share}\,\mathbf{1}. \tag{25}$$

Unlike checkout, rewind does not consume or return a complete Agent at the caller. The role authorizes one transition serialized with the child's unique complete writer. Descendant roles do not suffice for checkout, rewind, archive, give, get, or filtered receive.

B Dynamic Semantics

The small-step relation is

$$\langle p, e, \Xi, \Sigma \rangle \longrightarrow \langle p, e', \Xi', \Sigma' \rangle, \tag{26}$$

where p is the current participant and Ξ is the context map from Appendix A. The abstract state Σ contains the published resource states and the finite live parent–child relation T. Messages remain linear values in Ξ rather than a second store. Scheduling, transport, shell, and retry states are not components of the core configuration; they appear only in the runtime refinement.

The current participant determines the invoking context and where spawn! installs roles and provisioned permissions. Role pointers are derived from and checked against T; they do not reconstruct it. A reduction records only successful realization of the checked effect and atomically commits its post-contexts. Rejecting or abandoning a proposal before commit contributes no corresponding core transition. Whether a concrete operation realizes such a transition is determined by its committed effects; the runtime correspondence is described in Appendix E. External-world state is also absent from Σ, because explicit external operations remain classified as IO.

B.1 Atomic Evaluation

For every exact complete value $(a, s) : A$ held in Ξ, the published state for a agrees with s. For a reversible resource, agreement includes the same tree and current-node designation. The effect produced by checking is only an intermediate proposal: its successful realization precedes the joint commit of the corresponding contexts and published states.

The commit boundary is summarized by

$$\frac{\Xi; \Sigma\text{-view} \vdash_p J : C \dashv \Xi'; \mathit{effect} \qquad \mathit{effect} \text{ succeeds with } v, \Sigma'}{\langle p, J, \Xi, \Sigma\rangle \longrightarrow \langle p, v, \Xi', \Sigma'\rangle}. \tag{27}$$

Every primitive and privileged macro therefore appears as one atomic transition. The core semantics contains no configuration for an operation that is merely in progress or being retried; those are runtime refinement states.

B.2 Observation and Update

If an operation addresses a through $\mathsf{bundle}_A(R) : \mathsf{Bundle}_A(R)$, its successful reduction uses Rule 13: it installs $(a, s) : A$ and retains $\mathsf{bundle}_A(R - \{a\})$. The source of s is abstract; the rule requires only $s : \mathsf{State}_A$. No core configuration contains both the exact complete value at a and a complete bundle whose denotation still contains a.

If $a : \&\mathsf{File}$, then $\mathsf{read}(a)$ returns the current published $\mathsf{Payload}_{\mathsf{File}}$ and leaves Σ unchanged. For a reversible File it reads the label at the current node. If $(a, s) : \mathsf{File}$, then

$$\mathsf{write}((a, s), q) = (a, s') \tag{28}$$

atomically replaces the published state while preserving identity. For versioned state, s' adds one fresh node as a child of the current node, labels it with the new File payload, and makes it current. Edit has the same transition shape but is enabled only for a present current state containing the designated occurrence.

Listen preserves its participant role pointer and returns a kind-specific String observation. Reason returns a String for the current Human or Agent; if it changes that participant's published state, it does so in the same atomic step under the existing unique complete writer.

Send chooses a fresh message identity m and installs $(m, \mathsf{Msg}(q)) : \mathsf{Message}$ at the Human or Agent recipient. It transfers no pointer. Receive consumes every Message in the finite batch selected by $\mathsf{select}_p(h, \Xi)$ exactly once and returns its String encoding. A sender filter changes only this abstract selection and requires the ordinary pointer specified statically.

Steer creates a fresh Message under the target Agent's existing writer. Interrupt, stop, exec, and input have transitions only when their abstract availability predicates hold. Their scheduler effects, process-tree handling, delivery ordering, and concrete completion detection do not occur in the core configuration; Appendix E gives one implementation.

B.3 Participant Structure, Communication, and Ownership Transfer

When $p : Q$ spawns a fresh $a : P$, one transition creates the complete child, adds $p \to a$, installs direct-parent and direct-child roles at their endpoints, installs the descendant role at every pre-existing strict ancestor, and provisions selected permissions. Provisioning replaces the location of every selected complete value; it does not leave copies at both endpoints. This rule is independent of whether either endpoint is Human, Agent, or Term.

A successful $\mathsf{give!}((a, >), (b, t))$ removes $(b, t) : A$ from the parent and installs that same value at child a; child-to-parent give is symmetric through $(p, <)$. Parent-side $\mathsf{get!}((a, >), b)$ is enabled only when the direct child holds b and moves that value back to the parent. There is no observable transfer token, waiting claim, or in-transit owner. Competing transfers serialize, and deeper recovery is a sequence of direct-edge transitions. This is deliberately different from send: the latter copies a String into a fresh Message, while give/get move one existing complete value.

For a stopped direct child a, define the meta-level fold

$$\mathsf{Fold}_a(\Xi, T) = (\Xi_a, T_a) \tag{29}$$

recursively from the leaves. One recursive clause moves a leaf's complete external authority across its unique direct edge, consumes that leaf's participant and Message values, and removes the edge and every role it justified. The clause is repeated until the original edge into a returns the remaining external authority to the caller.

A successful archive! commits (Ξ_a, T_a) in one transition; the recursive clauses are not separate machine states. Thus every ownership movement used to compute the result is adjacent, while no intermediate fold is observable. Ordinary pointers outside the subtree remain ordinary pointers but cannot recreate an archived participant or a structural role.

B.4 Checkout, Rewind, and Atomic Term Commit

For a File state described by (V, E, r, c), checkout to $n \in V$ returns a complete File with the same identity, graph, root, and labels and with current node n. The published entry changes in the same step. The primitive does not replay a list of inverses or recreate an earlier complete File.

For a direct Agent child, rewind performs the corresponding current-node change under the direct-child role. It returns unit to the parent and never exposes the child's complete state. In both cases navigation leaves effects outside the selected tree unchanged: it does not alter Message liveness, completed give/get transitions, another resource's current node, or IO.

A successful Term operation may produce a finite update map $U = \{a_i \mapsto s_i'\}_{i \in I}$. If the Term holds the complete value for every a_i, the core semantics represents publication by the single abstract transition

$$(\Xi, \Sigma) \xrightarrow{\mathsf{commit}(U)} (\Xi[\,(a_i, s_i) \mapsto (a_i, s_i')\,]_{i \in I}, \Sigma[U]). \tag{30}$$

The identities and ownership locations are preserved, and no proper subset of U is published as an intermediate core state. Consequently, a read precedes or follows the entire commit and never observes a partial update.

The calculus does not model epochs, shell jobs, physical writes, snapshots, or sandbox rebuilding. Appendix E describes those states and the refinement condition by which their successful completion implements Rule 30. Functions used to implement checkout are subject to the analogous single-transition refinement.

C Soundness

C.1 Auxiliary Notions

Definition 1 (Well-formed core configuration). A pair (Ξ, Σ) is well formed, written $\mathsf{WF}(\Xi, \Sigma)$, when:

1. every entry of Ξ is well typed, and every linear value occurs in exactly one linear position;
2. $\mathsf{cov}_\Xi(a) \leq 1$ for every path atom a; for a non-path identity, at most one exact complete value has that identity;
3. the state published for every exact complete value $(a, s) : A$ agrees with s;
4. the live relation T in Σ is a finite rooted tree, and every direct-child, descendant, and direct-parent role in Ξ is held at a parent, strict ancestor, or child, respectively, that justifies it;
5. every reversible File or Agent state is a connected acyclic tree of payloads whose root and current nodes belong to that tree;
6. every version-tree payload is capability-free; and

7. every live participant has one stable kind and every newly allocated participant, resource, or Message identity is globally fresh.

For uniform notation, let $N_\Xi(a) = \mathsf{cov}_\Xi(a)$ for path atoms and let it count exact complete values for other identities. Ordinary and role pointers are never counted. Well-formedness gives $N_\Xi(a) \le 1$.

An operation J is available when the abstract side conditions of its rule hold. Examples include freshness for spawn, membership of a selected version node, possession of the selected value by get's direct child, and the stopped premise for archive. Lifecycle and external-execution conditions are left as abstract predicates: the core soundness results depend only on whether the premise holds at the transition, not on the implementation state that establishes it. Write $\mathsf{blocked}(J, \Sigma)$ when a well-typed exposed operation has no rule solely because such an availability premise is false.

C.2 Basic Type Safety

We use the standard linear substitution facts: unrestricted substitution preserves a linear context, while substituting a linear value consumes its one designated assumption. Disjoint context splitting is preserved by both forms.

Lemma 2 (Materialization conserves coverage). If $a \in R$, replacing $\mathsf{bundle}_A(R)$ by $(a, s) : A$ and $\mathsf{bundle}_A(R - \{a\})$ preserves $\mathsf{cov}_\Xi(b)$ for every atom b.

Proof. For $b = a$, the exact value replaces the bundle's one contribution. For $b \neq a$, membership in the residual bundle is unchanged. □

Lemma 3 (Archive-fold safety). If $\mathsf{WF}(\Xi, \Sigma)$ and $\mathsf{Fold}_a(\Xi, T) = (\Xi_a, T_a)$ is defined, the fold preserves authority coverage, moves every retained complete value only across adjacent edges, and leaves no role in Ξ_a justified solely by a removed edge.

Proof sketch. Induct on the height of the stopped subtree. A leaf clause moves or consumes each linear value once and removes exactly its incident edge and justified roles. Removing all leaves yields the induction hypothesis for the smaller tree; the final clause crosses the original direct-child edge. □

Lemma 4 (Preservation). Suppose $\Gamma_p; \Delta_p \vdash e : C$, $\mathsf{WF}(\Xi, \Sigma)$, and $\langle p, e, \Xi, \Sigma\rangle \to \langle p, e', \Xi', \Sigma'\rangle$. Then e' has type C in the resulting context of p, and $\mathsf{WF}(\Xi', \Sigma')$.

Proof sketch. Proceed by cases on the successful reduction. Pure cases use context splitting and linear substitution. Non-transferring observations preserve Ξ's linear entries. Send creates one Message at a globally fresh identity, while receive consumes every selected Message once. Native updates, checkout, and Rule 30 replace complete values by successors at the same identities and commit matching published states.

Materialization preserves every atom's coverage by Lemma 2. Give/get relocate one value across a direct edge. Spawn extends the forest with one fresh leaf, derives only roles justified by the new relation, and moves provisioned complete values. Archive uses Lemma 3. Rewind changes the current node of the already unique Agent; the remaining version cases use Lemma 13. Thus each component of $\mathsf{WF}(\Xi', \Sigma')$ is preserved. □

Lemma 5 (Relative progress). If e is closed and well typed in participant p and $\mathsf{WF}(\Xi, \Sigma)$, then one of the following holds: e is a value; the configuration can reduce; or evaluation exposes a well-typed operation J with $\mathsf{blocked}(J, \Sigma)$.

Proof sketch. Induct on the typing derivation. Standard elimination forms either reduce a proper subterm or expose an operation. If its abstract availability premises hold, Rule 27 supplies the successful atomic step; if one is false, the third case records that explicit boundary. A concrete attempt that ends without realizing a checked transition contributes no core step. Concrete failure reporting belongs to the runtime refinement in Appendix E. □

Theorem 6 (Type safety). A closed, well-typed program starting in a well-formed state cannot reach a type error through core reductions. Every reachable configuration is a value, can reduce, or is blocked only on a declared abstract availability premise.

Proof. Induct on the reduction sequence using Lemmas 4 and 5. □

C.3 Unique Ownership and Capability Confinement

Theorem 7 (Unique complete authority). For every identity a in every core configuration reachable from a well-formed initial configuration, $N_{\Xi}(a) \leq 1$.

Proof sketch. The initial bound is a clause of well-formedness, and Preservation maintains that clause. The nontrivial cases are explicit: materialization conserves coverage by Lemma 2; give/get and spawn provisioning relocate rather than copy a linear value; archive preserves or decreases coverage by Lemma 3; and fresh allocation introduces authority only at a previously unused identity. Updates, navigation, and finite Term commit preserve identities and ownership locations. □

Corollary 8 (At most one complete writer). Two computations cannot simultaneously hold complete values for the same identity. A parent invoking rewind receives no second complete Agent; the restricted transition is serialized with the child's writer.

Theorem 9 (No capability transmission by send). A successful send transmits no ordinary pointer, role pointer, or complete ownership.

Proof. Send's payload has type String, and there is no conversion from String to $\&A$, $\&_{>}P$, $\&_{>>}P$, $\&_{<}P$, or A. □

C.4 Role-Pointer Provenance

Theorem 10 (Role authenticity). In every reachable well-formed core configuration, $(a, >) : \&_{>}P$ implies that its holder is the recorded parent of a; $(a, >>) : \&_{>>}P$ implies that its holder is a recorded strict ancestor of a; and $(p, <) : \&_{<}P$ implies that its holder is a recorded child of p.

Proof sketch. Induct on core reductions. Spawn introduces direct roles together with a fresh edge and descendant roles only at its existing ancestors. Weakening preserves strict ancestry; contraction may duplicate an unrestricted role only within the same justified holder; and erasure produces an ordinary pointer. Send accepts only String, while transfer moves complete objects rather than roles. Finally, Fold removes every role whose justifying edge or reachability fact disappears. No other rule introduces a role or changes T. □

Corollary 11 (Authorized direct-edge macros). Every reducible rewind or archive is justified by an actual direct parent–child edge. Ordinary pointers and descendant roles are insufficient; descendant roles are also insufficient for give or get.

Corollary 12 (Participant-kind neutrality). Unique ownership and role authenticity apply equally to Human, Agent, and Term at every supported tree level.

C.5 Version and Atomicity Safety

Lemma 13 (Version-tree preservation). Checkout and versioned write preserve the complete File-state shape $\mathsf{State}_{\mathsf{File}} = \mathsf{Tree}^{\bullet\bullet}(\mathsf{Payload}_{\mathsf{File}})$; the corresponding argument holds for Agent state.

Proof sketch. Checkout leaves the graph unchanged and selects a node already in V. Write adds one fresh node and one edge to the old current node. Connectedness is preserved, and a cycle cannot arise because the fresh node has only that new incident edge. Root and current-node membership remain valid. □

Theorem 14 (Checkout locality and no ownership resurrection). Checkout and rewind change only the current-node designation of an existing version tree. They do not recreate a consumed complete object or reverse an effect outside that tree. In particular, for target a and every $b \neq a$, the transition preserves $\Sigma(b)$ and all ownership locations in Ξ.

Proof sketch. The two rules preserve the selected tree's node set, edges, root, and labels and replace only its current-node designation. Their conclusions do not update any other published entry or participant context. Moreover, payloads are capability-free by well-formedness, so selecting an old node cannot introduce a complete value or role. The result follows directly for every state component outside the singleton footprint $\{a\}$. □

Theorem 15 (Atomic observation). A read observes one complete published File payload. Relative to a native write or finite Term commit containing that identity, it returns either the preceding or resulting payload and never an intermediate state.

Proof sketch. A native update changes its complete value and published entry in one reduction. Rule 30 changes its finite update set in one reduction and publishes no proper subset. A read is one reduction consulting the published entry, so the execution order places it before or after the complete update. □

Concrete epoch and snapshot behavior is not a premise of this theorem. It is a runtime refinement obligation discussed in Appendix E.

These results establish the intended boundary without asserting liveness. Pointers permit persistent observation without duplicating update authority; path refinement replaces bundle membership by an exact resource and a disjoint residual bundle; structural operations preserve role provenance; navigation does not resurrect authority; and abstract resource transitions expose only complete states.

D Programming Language

D.1 Surface Form

The initial compiler pipeline is Source → AST → Checked IR; it requires neither an optimizer nor native code generation. The language uses imperative notation:

```
a : File
x = read &a
write a "first revision"

&>A = spawn! Agent "inspect the implementation" &a
send &A "Check the current implementation."
info = listen &>A
give! &>A a

states = log &>>A
n = choose states
rewind! &>A n
```

Updating a consumes the old complete binding and returns the successor under the same source name. The compiler rejects overwriting a still-live complete value. Give consumes the sender's binding; a later successful get or give-back may reintroduce it. Pointer operations do not consume the complete value. If the concrete tool fails, the post-context is not committed and the pre-operation binding remains live.

The metavariable selected by choose denotes a scoped existing state ID at the UI or tool boundary. It does not add a state-ID type or node resource to the core calculus. Pi reasoning remains in native session content rather than becoming an explicit authority-changing tool.

D.2 Front End and Static Checking

The parser retains source spans so diagnostics can identify the command that violates ownership, role, lifecycle, or path premises. The checker implements only the constraints in Appendix A. It tracks complete value moves and rebinding; permits non-consuming projection; rejects use after give; checks give/get, checkout/rewind, spawn/archive, and version-node scope; and treats Human, Agent, and Term uniformly before applying kind-specific operations.

Role values can arise only from the live participant relation. Descendant roles admit the common listen, wait, interrupt, and stop operations. Direct roles remain necessary for transfer, rewind, and archive. Human and Agent may send, receive, and reason; Agent additionally steers and rewinds; Term exec and input are lifecycle-gated. Agent and Term log follow the self/descendant rules in Rule 19. Send and receive preserve the String boundary and the one-consumption rule for each Message ID.

The compiler and runtime share one checking entry point. Its inputs are the persisted Γ_p/Δ_p contexts and a read-only view of published state, lifecycle, Message, participant-tree, and Term-epoch facts. A successful check produces a result type, complete proposed post-context, and Share/IO classification; it does not change the persisted contexts. Only after the concrete effect succeeds does the runtime commit that post-context together with the published state. The runtime does not reinterpret a secondary consumes/preserves/produces identifier summary.

Checked history stores typed identities, operation kinds, version or checkpoint identifiers, lifecycle/epoch transitions, and required blob or hash references. It does not duplicate write contents, message text, spawn prompts, commands, or stdin as a second typed information-flow history. Rewind and checkout select existing states rather than replaying those parameters.

D.3 Typed Items and Auto-Refinement

Tool arguments denote typed terms even when their surface representation is a String. Resource families use file: and web:; participants use human:, agent:, and term:. A displayed ID is a name, not a capability. Structural operations still require a live role derived from the participant tree.

A trailing slash distinguishes a File region from an exact File. File identities are canonical absolute paths within the immutable session base; relative forms and the home-directory shorthand are normalized at the boundary and do not become persisted identities. File existence is state, so an exact File may be Absent. Web identities are host-selector bundles; scheme, port, and URL path are not part of that identity.

An authority-bearing argument must use a typed reference. For a write or move, auto-refinement resolves the normalized target against the most specific held writable File region and partitions that region according to Equation 12. A read pointer follows set inclusion and is not shadowed by a more specific materialized File. Failure to resolve held authority is an error rather than implicit amplification. Complete grants use A and move ownership; projected grants use $\&A$ and leave the owner unchanged.

D.4 Compiler Validation

The compiler test suite contains positive and negative cases for: use after transfer; taking the same complete resource twice; overwriting a live linear binding; preserving a binding after a failed operation; writing through a read pointer; forged or wrong-direction roles; cross-level transfer; confusing a participant ID with a role; ready/stopped refinement; interrupt versus stop; archive of a non-direct or non-stopped participant; invalid Term lifecycle; non-atomic Term epoch lowering; out-of-scope checkout or rewind nodes; duplicate Message consumption; and attempts to place a capability where only String is accepted. Runtime integration proceeds only from a successful checked transition.

E Runtime Details

The runtime realizes the core context map Ξ and published state Σ through committed participant contexts, resource versions, and the live participant tree. A checked post-context remains a proposal until publication. Concrete execution steps that do not publish changes leave the corresponding core state unchanged; publication updates the affected contexts and resource states together in one core transition.

For example, while a Term modifies a File, its physical contents may already differ from the published version. Another Agent reading that File through Linden's native File interface receives the last committed payload from the pre-epoch snapshot, without waiting for the writer. Once foreground and background work has quiesced, reconciliation publishes the supported File changes together. Reads before this publication observe the previous payload; reads after it observe the new payload. The physical intermediate contents therefore do not appear as intermediate published states in the core semantics.

Command exit status is separate from publication. A command may exit unsuccessfully after modifying files, and those changes can still be reconciled and committed. Likewise, an error reported after publication does not reverse the committed core transition.

E.1 Authority Manager

Each session has one authority manager. Human interactions, Agent tools, Term sandboxes, native File operations, participant control, and ownership transfer all consult the same persisted Γ/Δ state. The compiler is the sole implementation of the authority judgment. The runtime derives a read-only Σ view, requests a checked post-context, performs the concrete adapter operation, and commits that post-context atomically. Indexes, caches, and sandbox policies realize this state but never become independent authority truths.

E.2 Resource Gateways

Native read, stat, write, and edit resolve typed File references through the manager. Read and stat observe the committed Absent/Present state through held set-valued pointers. Write and edit require the resolved complete atom, perform the physical operation, and publish the checked successor only on success. Canonicalization, traversal checks, symbolic links, hard links, and private runtime paths are physical-enforcement obligations; none changes the calculus's authority algebra.

File regions and Web host selectors realize the path sets from Section 3.1. Refinement materializes or extracts atoms and records exclusions so that the residual descriptions remain disjoint. Network adapters additionally enforce the selected host scope and reject private, loopback, link-local, reserved, mapped, or rebound destinations as specified by the runtime policy.

E.3 Sandboxed Term

Term is a participant, while its controlled shell and sandbox are its adapter. Exec is enabled only when the Term is ready; input only while its foreground command is running. Foreground completion comes from shell execution state, not from matching a displayed prompt. Background quiescence combines shell job state with sandbox-level process supervision so detached work is not mistaken for completion.

The adapter builds each epoch from a fixed authority snapshot and preserves pre-epoch File snapshots for overlapping native reads and stats until reconciliation publishes the multi-resource transition described above. Authority changes that conflict with the live sandbox wait for this boundary. Live listen output is bounded and ephemeral; completed epoch/command metadata and bounded trace ranges back Term log. A recovered incomplete epoch is marked as such rather than reconstructed from missing output.

E.4 Participant Mechanism

The participant registry stores stable kind, parent, lifecycle, and adapter state for Human, Agent, and Term. The live relation is the structural source of truth: roles are derived from and checked against it at the operation boundary, and discarding a role does not remove an edge. Common listen/wait/interrupt/stop behavior is kind-neutral; archive remains a stopped direct-child transition whose internal subtree fold proceeds from leaves across adjacent edges. Give and get move authority only along direct edges and serialize through the same writer gate used by resource updates. A running owner may be interrupted when the operation requires quiescence, but get is not silently expanded into recursive stop.

Human and Agent share the Message representation. Ordinary follow-ups enter the persisted mailbox and are consumed in FIFO batches; steering uses its priority delivery class. A delivery claim is confirmed only when its custom message begins, so recovery can reopen an unconfirmed claim rather than silently drop it. Runtime notices explain context or recovery changes but are not Messages, do not enter the mailbox, and never move authority.

The current Human UI is an adapter to this participant mechanism. It may hold complete resources, pointers, and roles and may invoke the same checked operations as its type permits. It does not bypass the authority manager. Other Human scheduling adapters remain outside the present implementation and outside the core calculus.

E.5 Pi Session Tree, Rewind, and Log

Pi's session tree is the only Agent context history. Linden associates typed Agent checkpoints with stable Pi entries instead of copying reasoning, messages, tool inputs, or outputs into another context tree. Agent log renders this tree with raw entry IDs, the nearest typed state, and scoped rewindability. Raw IDs remain Strings and do not grant access to another Agent's tree.

Rewind first quiesces the selected direct child, moves its Pi context to the chosen entry, and reconciles the checkpoint with current session authority. Branch summaries are disabled for exact rewind because they would import information from the discarded suffix. The runtime then appends a notice under the selected node listing the target, effects not reverted, and current authority, and associates a reconciled checkpoint with the new leaf. Rewind does not automatically initiate a new model turn. Continued work requires a later Human/parent message or steer operation.

File checkout remains a distinct resource operation. Rewinding an Agent never changes a File's current version, and checking out a File never changes the Agent context. This separation realizes the local-navigation theorem rather than an implicit global rollback.

F Evaluation Details

F.1 RQ1: Semantic Enforcement

Test construction. Tests construct participant hierarchies and resource assignments, execute operations through the checker or runtime, and compare the resulting state with the expected transition. Assertions inspect complete ownership, role validity, resource versions, participant state, and durable records. Rejection tests compare state before and after the attempted operation.

The complete suite contains 117 tests. Its property-focused coverage comprises the following 28 tests; all pass.

Property	Tests	Principal assertions
Linear ownership and transfer	4	Complete ownership remains unique across writes and transfers; former owners lose write authority.
Role authenticity and structural control	4	Roles derive from participant relationships; resource transfer follows direct parent–child edges.
Path and private-state isolation	4	Traversal, symlink escapes, hardlink aliases, and access to runtime-private state are rejected.
Revision and effect persistence	1	Agent checkpoints are resource-scoped, and rewind preserves completed ownership transfers.
Failure atomicity	4	Rejected judgments, invalid transfers, failed creation, and excessive-depth spawning leave the relevant state unchanged.
Persistence and recovery	3	Restart recovers prepared writes, pending message delivery, and the active participant set.
Physical enforcement	8	Sandboxed execution respects filesystem and network authority and reconciles File versions.

Physical-boundary tests. These tests execute real Bash processes inside Bubblewrap. They verify network denial without Web authority, isolation of host-private configuration, and writes restricted to held Path authority. A delegated exact File is over-mounted read-only inside its former owner's broader Path, testing the interaction between coarse and fine-grained grants.

The remaining cases exercise versioning and concurrent filesystem effects: reads during a long-running Bash write return the last committed snapshot; deleting an owned File records an Absent state; modifying an absent-File placeholder records divergence; and a placeholder-reservation race preserves an external replacement. Together, these tests connect authority-state transitions to observable process and filesystem behavior.

F.2 RQ2: Runtime Measurement

Benchmark coverage. The 40 cases comprise read/stat/authority queries (7), write/edit operations (6), ownership transfer and messaging (4), checkpoint/checkout/rewind (4), ledger persistence (4), Bash execution (2), logical spawning (3), hosted spawning (2), and depth checks (8).

Measurement procedure. Inexpensive cases collect 200 samples after 20 warm-up samples. Expensive cases collect 30 samples after three to five warm-ups. Short queries batch multiple operations within a sample, and elapsed time is normalized per operation. Independent comparisons use randomized blocks; causally linked operations, such as give/get and send/receive, retain their execution order.

Measurements use Node.js and Bubblewrap under Linux/WSL2, with Linden and native baselines accessing the same Windows-mounted filesystem. The timed region covers the operation after runtime initialization. Bash measurements start a new sandbox for each invocation. Logical spawning creates participant and authority state; hosted spawning additionally creates a persisted Pi session, with model invocation disabled.

Baselines and latency accounting. Native write, edit, and content-restoration baselines use atomic replacement and fsync. Added latency is calculated as Linden p50 minus native p50. The main text reports common file and Bash operations; the following table provides complementary state-management and participant-creation measurements.

Operation	p50 (ms)	p95 (ms)
Give	2.04	2.39
Get	2.05	2.42
Send with durable payload	4.33	5.45
Receive	2.70	3.80
Agent checkpoint	1.92	2.53
Agent rewind	1.88	2.46
File checkout	5.69	6.71
Logical spawn, top-level	2.53	3.15
Logical spawn, nested	2.49	3.63
Hosted spawn, top-level	14.94	16.02
Hosted spawn, nested	12.80	15.91

File checkout adds 4.11 ms over native durable restoration, whose median is 1.58 ms. Rejecting an excessive-depth spawn takes 3.69 μs at p50 and 4.23 μs at p95. Read-authority checks take approximately 0.26 μs at each tested participant depth—1, 2, 3, and 5.

F.3 RQ3: Task Utility and Dynamic Delegation

Experimental conditions. Tasks are validated using the official blank-environment and reference-solution controls. Each condition contains 100 task–repetition trials, with two repetitions per task. All trials use gpt-5.6-luna with max reasoning. Task prompts and task-specific deadlines are shared across conditions; deadlines range from 12.5 to 60 minutes.

Condition	Execution interface
Pi-Naive	Native Pi tools.
Sandbox	Pi tools with physical sandboxing.
Linden	Linden authority management, checked tools, versioned files, and Terms; additional Agent creation disabled.
Linden + Delegation	Linden facilities with additional Agent creation enabled.

Both Linden conditions begin with one task-solving Agent. With delegation enabled, the model chooses whether to spawn additional Agents, their assignments, and their resource grants. Descendants share the trial's remaining deadline and contribute to its aggregate usage.

Outcome accounting. Each trial records its grader outcome separately from execution status. A candidate produced before a timeout can subsequently pass the grader. Trials without a usable grade remain unscored. Usable grades are available for 98 Pi-Naive, 98 Sandbox, 89 Linden, and 85 Linden + Delegation trials out of 100 per condition. The main-text comparison uses the intersection of graded trials: 74 task–repetition pairs shared by all four conditions. Success retention is the proportion of Linden successes also solved by Linden + Delegation: 42 of 46, or 91.3%. Complementary completion is the proportion of Linden failures solved by Linden + Delegation: 8 of 28, or 28.6%. Four pairs show the reverse outcome, with Linden succeeding and Linden + Delegation failing.

Execution and grading records. Trials affected by experimental infrastructure failures were resumed to complete the evaluation records. Saved candidates were graded where possible to recover missing scores.

Authority-boundary failures. We count a failed tool call when its recorded tool result has isError=true; model-network retries are excluded. The counts below use the 74 task–repetition pairs in the main comparison. Session traces are available for every condition except Linden + Delegation, where four wrapper timeouts leave 70 pairs with readable tool results.

Condition	Logged	All failed tool calls	Boundary-related failures
Pi-Naive	74	184	1 OS-level privilege failure
Sandbox	74	292	16 static-authority denials; 26 OS/sandbox permission failures
Linden	74	179	9 unauthorized file accesses
Linden + Delegation	70	215	87 role-topology rejections; 19 unauthorized file accesses; 8 unauthorized Bash working-directory starts

Sandbox failures commonly arise during temporary-file use, package installation, and service setup; a denied operation can consequently surface later as a missing dependency, program exception, or timeout. In Linden + Delegation, role-topology rejections occur primarily when an Agent applies ancestor-authorized lifecycle operations such as wait or listen to a parent, sibling, or otherwise unrelated participant. Direct peer messaging remains available, but lifecycle control is not peer-symmetric. Unauthorized file accesses similarly reject operations beyond a descendant's delegated resources.

These counts diagnose failure modes rather than normalize execution cost: delegation produces more tool calls, and a command launched inside a persistent Term can exit nonzero without its successful exec launch being recorded as a failed tool call.

Delegation metrics. Runtime metrics are available for 70 of the 74 paired Linden + Delegation trials. Additional-Agent count measures cumulative creation during a trial, including Agents that subsequently stop. Depth counts the initial Agent as level one and follows Agent ancestry; Terms are counted separately.

Across these trials, 47 create additional Agents, totaling 166 additional Agents. The median additional-Agent count across all 70 trials is two, and the maximum is 38. Among delegating trials, 40 create one to three additional Agents, six create four to nine, and one creates 38. Maximum observed depth is two in 41 delegating trials, three in four, and five or nine in one each.

The Linden interface retains Term creation through the shared spawn tool. Agent-creation requests receive a disabled-operation response; such requests are included in tool-use accounting, while creation counts include successful creation only. Terms are created in 25 of the 74 paired Linden trials.

Resource accounting. Within the 74-pair comparison, usage records are available for all Linden trials and 70 Linden + Delegation trials. Aggregate token usage includes all recorded Agent sessions and cached tokens. The 70 pairs with usage records in both conditions yield median per-pair delegation-enabled/delegation-disabled ratios of 2.32 for tokens, 1.73 for recorded estimated cost, and 2.60 for tool calls.

F.3.1 Task-Level Analysis

The following traces detail the operations underlying the examples in the main text.

Delegated implementation analysis. The interpreter task requires executing a supplied MIPS binary and producing a rendered frame. The parent creates three read-only children to inspect the executable, its source, and runtime requirements. Their reports identify three implementation constraints:

- The executable uses a custom zero-based syscall convention.
- Its approximately 1 GB BSS region favors sparse memory allocation.
- Execution requires MIPS32r2 instructions and delay-slot handling.

The parent implements ELF loading, sparse memory, instruction execution, and custom syscalls. The resulting candidate passes all three grader checks. In the other repetition, Linden + Delegation completes the task with the initial Agent alone.

Recovery of a deleted input. The database-recovery task supplies a database and an encoded write-ahead log. In one repetition, a SQLite probe returns five rows, after which the WAL file is absent. Its version history contains both the original contents and the subsequent Absent state recorded around Bash execution. The Agent checks out the original version, makes a backup, decodes the WAL using XOR key 0x42, and verifies recovery of 11 records in a copied database before writing the requested JSON. Both repetitions use version history to restore the WAL and pass all seven checks.

Persistent inference service. The model-inference task requires an API that remains available for grading. All four Linden trials use Terms with a single Agent and pass all four checks. In one Linden + Delegation trial, the Agent starts the service in a Term and uses execution and listening operations to manage it. In the second Sandbox repetition, model download succeeds, while the three API checks report connection-refused errors. The contrast identifies service availability at grading time as the relevant execution outcome.

F.4 RQ4: Multi-Agent Collaboration

Conditions and grading. We use the Flash subset of CooperBench and run every model role through a Pi AgentSession. Every model role in every condition uses the same model and reasoning level. Conditions also share the task image, global budget, deadline, compression settings, and total container resources. The Pi and Linden conditions use one shared task workspace. Shepherd retains its separate worker workspaces and supervised patch-integration procedure. Each condition ultimately submits one joint patch; the same patch is applied in a fresh grader for each feature. A trial succeeds only when every feature retained by the scoring protocol passes.

Condition	Initial topology	Model Agents	Collaboration rule
Pi-Solo	$H \to W$	1	One Agent implements both features.
Pi-Coop	$H \to \{W_1, W_2\}$	2	Equal peers implement one feature each.
Pi-Team	$H \to L \to W$	2	A lead-worker and member implement one feature each.
Linden-Coop	$H \to \{W_1, W_2\}$	2+	Equal peers coordinate typed authority through H.
Linden-Team	$H \to M \to \{W_1, W_2\}$	3+	A pure manager coordinates two feature workers.
Shepherd	$S \to \{W_1, W_2\}$	3	A supervisor observes two isolated workers and integrates patches.

The Human in Linden-Coop is a deterministic, non-model authority broker. It starts both peers behind a barrier and processes authority requests in FIFO order, but does not inspect code, allocate tasks, or repair deadlocks. Peers communicate directly. Sibling ownership movement follows $W_1 \to H \to W_2$ through ordinary give/get transitions. Linden-Team instead gives a model manager both feature descriptions and initial workspace ownership; the manager transfers authority to its workers. Additional Agent creation is available in both Linden conditions and is charged to the shared trial budget. It is unused by Linden-Coop on the reported set and occurs in four Linden-Team trials, creating five additional Agents in total.

Common valid set. The six conditions have valid formal grades on a common intersection of 31 Flash feature pairs, each run twice. This intersection is determined by grade availability rather than outcome.

A separate specification audit identifies criteria for which the task text is ambiguous or inconsistent with the supplied tests. Table 2 lists the resulting global exclusion set. Exclusions are applied identically to every condition and do not alter any remaining feature grade. A trial passes when all of its remaining features pass. If both features are excluded, the trial contributes to neither numerator nor denominator. This removes four task–repetition trials per condition and leaves 58 trials per condition.

Results. Table 3 reports the audited score used in the main text. Linden-Team ranks first on the audited set. In paired comparisons, Linden-Coop has four unique successes

Table 2: CooperBench Flash feature criteria excluded by the specification audit. The same set is used for every condition.

Task	Features	Reason
huggingface_datasets/task3997	2, 4	underspecified behavior
llama_index/task18813	5	ambiguous boundary semantics
openai_tiktoken/task0	9	task/test naming mismatch
pallets_click/task2800	2	ambiguous exception semantics
pallets_jinja/task1559	8	unspecified fallback behavior
pallets_jinja/task1621	4	unspecified invalid-input behavior
pillow/task25	5	ambiguous mutation semantics
pillow/task68	5	unspecified exact log message
pillow/task290	3, 5	ambiguous factor boundaries
dirty_equals/task43	3	unspecified email/TLD range

to Pi-Coop's two; Linden-Team has four unique successes to Pi-Team's one and three to Shepherd's two. Relative to Linden-Coop, Linden-Team adds two successes without losing any shared success.

Table 3: CooperBench Flash results on the 58-trial common valid set after the global exclusions in Table 2.

Condition	Audited success
Pi-Solo	45/58 (77.6%)
Pi-Coop	48/58 (82.8%)
Pi-Team	49/58 (84.5%)
Linden-Coop	50/58 (86.2%)
Linden-Team	52/58 (89.7%)
Shepherd	51/58 (87.9%)

Integrating overlapping changes. The main-text example uses features 2 and 9 from the Jinja loader task pallets_jinja_task/task1621. Feature 2 adds path normalization and feature 9 adds automatic cache invalidation; both modify src/jinja2/loaders.py. In both repetitions, Pi-Team's lead treats the member's cache-invalidation implementation as outside the requested scope and directs its removal. The resulting joint patches pass feature 2 and fail feature 9.

In Linden-Team, the manager retains both feature descriptions and makes the shared file handoff part of the work plan. It gives the file to the normalization worker, receives it back, and then gives it to the cache-invalidation worker. Both final patches retain the two implementations and pass both feature graders. The trace illustrates the complete coordination design—a shared task view together with typed resource handoffs—rather than isolating either component as a single-factor effect.

F.5 RQ5: Large-Scale Multi-Agent Collaboration

Benchmark and conditions. Silo-Bench (Zhang et al., 2026) distributes task input across private Agent shards and tests whether the group can compute an answer from distributed information. Its 30 algorithmic tasks cover aggregation, local dependencies, and global reorganization. We evaluate four communication conditions at $N \in \{2, 10, 50, 100\}$ model Agents:

Condition	Coordination mechanism
Silo-P2P	Original round-synchronous point-to-point protocol.
Silo-Broadcast	Original round-synchronous broadcast protocol.
Silo-SFS	Original round-visible shared-storage protocol.
Silo-Linden	Linden participant discovery, direct messaging, and typed resource authority under native asynchronous execution.

The Linden topology contains one non-model Human broker and exactly N model Agents. Each Agent initially receives only its private shard. The broker does not plan, compute,

aggregate answers, or relay ordinary messages; it only performs deterministic authority transitions. Additional model-Agent spawning is disabled so that N remains fixed. Agents may deliberately share shard information through the condition's communication mechanism, but cannot read another Agent's private state directly.

Comparison and metrics. Within each task instance and scale, all four conditions use the same generated inputs, shards, model, task instructions, global token protection, and wall-clock deadline, while retaining their protocol-specific communication surfaces. The three original conditions retain Silo-Bench's round-synchronous semantics, whereas Linden retains its asynchronous runtime. The experiment therefore compares complete coordination environments rather than isolating one messaging primitive.

The primary metric is complete task success. Secondary metrics include the fraction of correct Agent outputs, tokens, model calls, cost, wall-clock time, message count and volume, realized recipients, shared-storage operations, and Linden authority requests, waits, and denials. The experimental unit is a task instance, not an individual Agent. Results are reported separately by task family and Agent count so that aggregate scores do not hide scale-dependent coordination failures.

Execution status. The frozen experiment contains 480 trials: 30 tasks, four Agent counts, and four conditions. Of these, 153 finish normally with a valid evaluation, 275 fail, and 52 reach the wall-clock deadline. Table 4 gives the status by condition.

Table 4: Final status of the frozen Silo-Bench trials.

Condition	Completed	Failed	Timed out	Total
P2P	37	79	4	120
Broadcast	40	66	14	120
SFS	38	66	16	120
Linden	38	64	18	120
All	153	275	52	480

Only normally completed trials receive a formal task score. Failed and timed-out trials arise from a mixture of budget exhaustion, model or connection errors, deadlines, and infrastructure failures. We therefore keep them out of the primary paired correctness comparison. As a complementary end-to-end measure, we also report an all-initiated view in which every such trial counts as unsuccessful; this view measures the probability that an initiated trial produces a fully correct result without attributing every failure to an incorrect answer.

Paired success and efficiency. The primary comparison uses the intersection of task–scale instances that complete in all four conditions. This produces 25 paired instances: 20 at $N = 2$ and five at $N = 10$. Table 5 reports full-task success, the per-instance fraction of correct Agents averaged across the set, and medians of the recorded token and execution-time measurements.

Table 5: Results on the 25 Silo-Bench instances completed by all four conditions.

Condition	Full success	Correct Agents	Median tokens	Median time (s)
P2P	4/25 (16%)	40.8%	41,709	282.9
Broadcast	10/25 (40%)	62.4%	30,606	254.1
SFS	12/25 (48%)	67.6%	34,579	254.2
Linden	19/25 (76%)	78.0%	50,136	87.2

Every paired instance solved by P2P, Broadcast, or SFS is also solved by Linden. Relative to SFS, Linden retains all 12 SFS successes and completes seven additional instances. Linden records more tokens at the median, while its asynchronous runtime completes the paired instances substantially faster than the round-synchronous conditions. Token totals follow

each runner's recorded accounting, including its cache accounting, and therefore describe the complete evaluated environments rather than a normalized model-only cost.

Table 6 separates the paired result by Agent count. All four Linden successes at $N = 10$ occur on instances for which none of the three baselines achieves full-task success.

Table 6: Full-task success in the common completed set by Agent count.

Scale	Instances	P2P	Broadcast	SFS	Linden
$N = 2$	20	4/20	10/20	12/20	15/20
$N = 10$	5	0/5	0/5	0/5	4/5

The paired set contains 13 Level-I aggregation instances, nine Level-II local dependency instances, and three Level-III global-reorganization instances. Table 7 gives the corresponding full-task outcomes.

Table 7: Full-task success in the common completed set by Silo-Bench level.

Level	Instances	P2P	Broadcast	SFS	Linden
I: Aggregation	13	2/13	6/13	7/13	12/13
II: Local dependencies	9	1/9	2/9	3/9	5/9
III: Global reorganization	3	1/3	2/3	2/3	2/3

There is no four-condition completed intersection at $N = 50$ or $N = 100$. We therefore use those scales for execution and trajectory evidence rather than estimating a paired success-rate difference from unmatched runs.

All-initiated success. Table 8 gives the complementary end-to-end view. Each condition contributes all 120 initiated trials, and failed or timed-out trials count as unsuccessful. This common denominator incorporates execution reliability while retaining full-task success as the numerator.

Table 8: Full-task success across all initiated Silo-Bench trials, overall and by task family.

Scope	Trials/condition	P2P	Broadcast	SFS	Linden
All tasks	120	4/120	13/120	13/120	27/120
I: Aggregation	40	2/40	8/40	8/40	14/40
II: Local dependencies	40	1/40	2/40	3/40	9/40
III: Global reorganization	40	1/40	3/40	2/40	4/40

The advantage is concentrated in aggregation and local-dependency tasks. On global reorganization, Linden succeeds on 4/40 trials versus 3/40 for the strongest baseline; global data exchange and execution completion remain the dominant limitations.

Results outside the paired intersection. For completeness, Table 9 reports every normally completed trial. These rows are not directly comparable because the completed task–scale instances differ across conditions, particularly at $N = 50$ and $N = 100$.

Terminal snapshots of censored runs. Failed and timed-out artifacts retain some submissions made before execution stops. We analyze these as a separate censored-at-stop snapshot. Across the 327 censored trials, 441 of 17,590 expected Agent submissions are present and 133 are completely correct. Table 10 reports submission coverage, correct submissions as a fraction of all expected Agents, and the continuous partial-quality score P defined by the frozen Silo evaluator. Unsubmitted Agents contribute zero to all three quantities.

These snapshots are not merged into formal success rates, but they show that Linden preserves more usable progress when an execution ends before every Agent reaches a terminal submission.

Table 9: Scores across all normally completed Silo-Bench trials.

Condition	Completed trials	Correct Agent outputs	Full-task success
P2P	37/120	212/438 (48.4%)	4/37 (10.8%)
Broadcast	40/120	363/516 (70.3%)	13/40 (32.5%)
SFS	38/120	299/496 (60.3%)	13/38 (34.2%)
Linden	38/120	266/400 (66.5%)	27/38 (71.1%)

Table 10: Terminal snapshots from failed and timed-out trials.

Condition	Submission coverage	Correct/expected	Partial quality P
P2P	0.41%	0.05%	0.05%
Broadcast	2.44%	0.02%	0.24%
SFS	2.34%	0.32%	0.72%
Linden	4.82%	2.60%	2.99%

Trajectory audit. In Any Match at $N = 10$, only Agent 8's private shard contains the matching error token. Agent 8 sends the positive finding to all nine siblings, and several recipients relay it. All ten Linden Agents submit the correct result; SFS reaches 8/10, P2P 6/10, and Broadcast 4/10. The Human controller relays no messages and grants no authority in this run: the evidence moves through ordinary role pointers while every private shard remains separately owned.

Pipeline Hash at $N = 100$ exercises a longer dependency structure. All 100 Agents submit the correct five-element segment. The sessions contain 357 direct-send calls, of which 346 succeed; 310 target an adjacent peer, covering 97 of 99 successor links and 98 of 99 predecessor links. Sixty Agents explicitly receive a result from their predecessor. When a predecessor is delayed, Agents may instead recompute the prefix from the genesis block: 54 session traces contain such a full-prefix computation, including 26 that also receive the predecessor result. The Human again relays no messages and grants no authority. The trace therefore combines a chain-shaped communication overlay with redundant local recovery while preserving independent authority over the Agents' private resources.

G Collaboration Case Studies

G.1 Bounded Write Delegation for Inference Batching

Task and resource allocation. The llm-inference-batching-scheduler task requires one plan for each of two 800-request input buckets. Each plan assigns every request exactly once to a batch and a compatible tensor shape. The two plans must jointly use at most eight distinct shapes and satisfy bucket-specific cost, padding, and latency limits. The parent creates two empty staging directories and spawns bucket-1-planner and bucket-2-planner, transferring one directory region to each. Both children inherit read access to the input files and cost-model script, but neither receives write authority over the input or final-output directories.

Child execution and handoff. Each child's first authority query confirms its assigned effective write region. Both use Python through bash to produce their staged JSONL files. B_1 produces 800 records in 333 batches (88,800 bytes); B_2 produces 800 records in 120 batches (89,600 bytes). The parent waits for the children to stop and invokes get on each directory. The successful results name both the returned region and its newly materialized plan File. The final authority snapshot places these resources at the parent and leaves neither child with File ownership. The trace therefore contains exactly two spawn-time grants followed by two explicit gets.

Validation and repair. The parent first copies both staged files byte-for-byte into its final-output directory. Validation accepts B_1 but finds that B_2 exceeds two limits (Table 11). The

parent regenerates only the official B_2 copy using a different grouping, yielding 197 batches. The staged original remains unchanged. Final joint validation checks request coverage, input hashes, per-batch shape consistency, disjoint batch identifiers, the global shape cap, and all four metric limits. It reports 530 total batches and two global shapes: $(2048, 32, 4096)$ and $(704, 32, 4096)$.

B_2 metric	Staged plan	Final copy	Required bound
Cost	4.8616×10^{10}	4.6406×10^{10}	$< 4.8 \times 10^{10}$
Padding ratio	0.16052	0.14620	< 0.15
P95 latency (ms)	209211.596	192365.8	< 210000
Sequential timecost (ms)	19564894.806	30347120.1	< 32000000

Table 11: Cost-model metrics before and after the parent's repair. These are metrics of the generated batching plans, not agent execution times. The repair satisfies all bounds; it does not improve every metric individually.

The final B_1 file matches its staged original byte-for-byte; the final B_2 file does not, reflecting the recorded parent repair. The retained originals separate child-produced artifacts from the parent's integration work. The saved run status is timed_out, while the final artifacts pass all six official verifier tests; artifact validity and run termination are distinct observations.

G.2 Recursive Read-Only Analysis for SQLite Recovery

Task and evolving tree. The task is to recover rows from a truncated SQLite file and write a JSON result. Its input and output paths are /app/trunc.db and /app/recover.json. The parent creates two independent analysts, db-recovery1 (D_1) and db-recovery2 (D_2). The latter then creates three children of its own: sqlforensic (F), manualparse (M), and hexinspect (X). Figure 2(c) shows the resulting six-Agent subtree. The second level is created by a child during execution, not by a parent preallocating five peer workers.

Shared observation and upward synthesis. The analysts inspect the same 4096-byte input using SQLite utilities, binary inspection, and Python parsers. Their reports identify a surviving table-leaf B-tree page without the usual database header and decode ten records using cell offsets and SQLite serial types. Manual parsing and byte-level inspection cross-check the integer and floating-point values. In the recorded message flow, manualparse and hexinspect report to their direct parent, db-recovery2, which sends its findings upward; db-recovery1 separately reports to worker. This describes the observed message flow; Linden also permits information messages to bypass direct tree edges.

The final authority snapshot gives all five descendants read pointers covering /app/ but no complete File values. Worker retains File authority and uses its own Python decoder to produce the ten-row JSON output. The recorded write reports file:/app/recover.json as the changed resource. A later read by db-recovery1 observes that output without acquiring ownership. The trial finishes with status completed and passes its official verifier test.

Structural control. The trace also records db-recovery2 attempting to wait on worker's ready state. The runtime rejects this with ROLE_MISMATCH: the child is not an ancestor of its parent. The same child can send findings to that parent, but possessing its identifier does not confer lifecycle-control authority. No explicit give, get, or rewind call occurs in this trace; its distinctive pattern is recursive sharing of observation with a single output owner, complementing the write delegation in Appendix G.1.