ASTER: CLOSING THE CONTROL GAP VIA THE AGENT-STATE TRANSACTIONAL EXECUTION RUNTIME

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent systems accumulate increasingly rich structure across model calls—memories, plans, resources, observations, and execution histories—only to flatten it into context when the next decision arrives. Once flattened, that structure can tell the model what is connected, but it cannot make those connections operational: every new event again relies on model inference to determine which consequences follow, how far they propagate, and what prior state should survive. We call this gap between represented structure and executable state change the **control gap**. Starting from this observation, **ASTER** (*Agent-State Transactional Execution Runtime*) takes a different path: rather than treating structured relations only as context for model inference, it makes them executable at runtime. ASTER represents committed state as a versioned typed hypergraph whose relation contracts derive the scope of change and successor conditions for each event. The model proposes a transition, while ASTER certifies its realized effects before atomic commit. Across nine external benchmarks spanning single-agent, multi-agent and visually grounded embodied tasks, interactive service and web environments, and code generation, ASTER outperforms popular baselines in all settings. ASTER can also plug into an existing agent harness as its planning-state runtime, showing that its transition semantics are not tied to a particular actor loop.

1 INTRODUCTION

Language-model agents increasingly maintain persistent structured state across model calls, including memories (Park et al., 2023), reusable skills (Wang et al., 2024; 2025), structured plans (Shen et al., 2023; Kim et al., 2024), and execution records. Such state allows later decisions to build on prior execution and has become a basic ingredient of long-horizon, interactive, and multi-agent systems.

Much of this progress addresses what we call the *context gap*: state relevant to a decision may not be available in the model's context when that decision is made. Memory retrieval, task graphs, checkpointing, and context management increasingly reduce this gap by preserving and exposing persistent state across model calls. Intriguingly, making structured state available to the model does not make the semantics of that state operational. The model is still relied upon to infer which consequences follow from a runtime event, how far those consequences propagate, and which previously committed state remains valid. We call this distinct problem the *control gap*: the gap between representing state structure and making that structure govern state change.

Figure 1 provides an illustrative example. On ParaCook, adding an explicit task graph to a strong model does not by itself improve task success over chain-of-thought. More importantly, even when the complete committed state is available in context, the resulting transition can still be inconsistent. In the breakfast example, coffee and toast are complete, an egg is frying while holding the pan, and final assembly depends on all three. If the egg is skipped, the resulting state needs to reflect the stopped action, the released pan, and the changed assembly requirement, while retaining the completed coffee and toast. Updating too little leaves stale consequences; updating too much discards valid prior work. Thus, richer context may help the model infer a better transition, but does not determine which transition the runtime accepts.

This problem arises throughout execution, not only during failure recovery or plan repair. Persistent agent state evolves whenever actions succeed or fail, observations become stale, resources are acquired or released, users change requirements, or peer agents complete shared work. Each event

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.

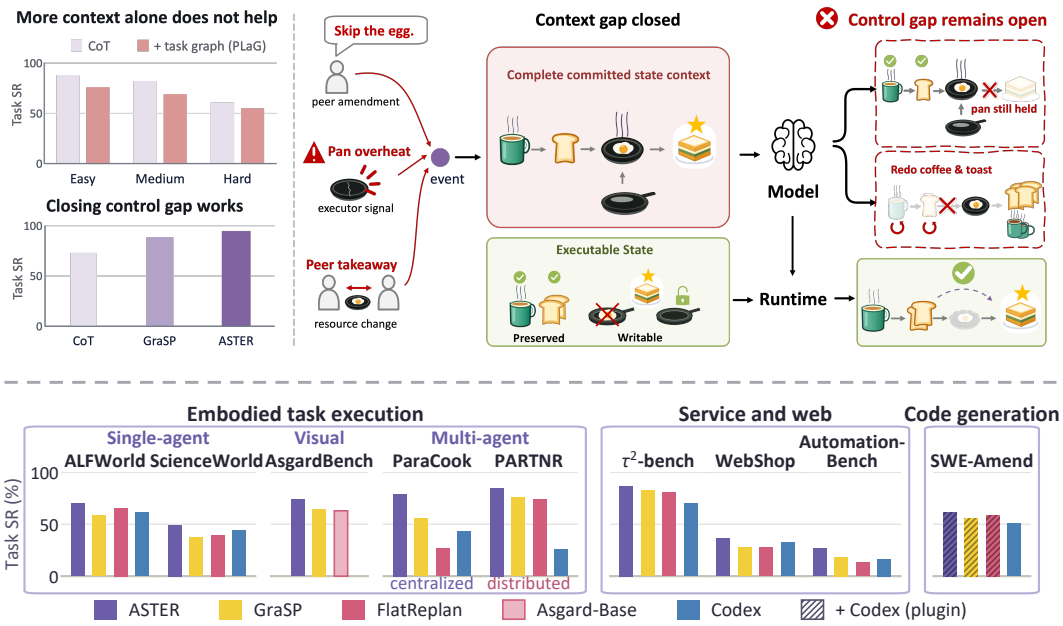


Figure 1: Closing the context gap does not close the control gap. *(Top left):* Published ParaCook results show that adding an explicit task graph to GPT-5 does not improve over chain-of-thought (Zhang et al., 2026). *(Top right):* Under a matched ParaCook setup, CoT, GraSP, and ASTER use the same tasks, GPT-5 (`high`), and episode budget. When a peer amendment Skip the egg., executor signal, or resource change alters the same breakfast, model-only revision either omits required consequences or rewrites unaffected work, even with the complete committed state in context. ASTER incorporates the event while preserving unaffected state, allowing the task to complete. *(Bottom):* Across the nine primary benchmark settings, ASTER achieves higher task success than FlatReplan, GraSP, and Codex.

can invalidate some committed state while leaving other state valid. *How can the structure maintained by an agent become executable semantics that govern the evolution of its committed state?*

To answer this question, we begin with a simple observation: a valid transition needs to incorporate the consequences induced by an event while preserving committed state that remains valid. We formalize these two requirements as *transition integrity*, consisting of *incorporation* and *preservation*. Importantly, connectivity alone is insufficient to determine how a change should propagate. Its consequences depend on both the relation type and the current execution state. For example, losing one prerequisite invalidates an *all-of* requirement, while an *any-of* requirement may remain satisfied by another alternative. Similarly, cancelling a running task may require releasing a resource it holds, whereas a completed result may remain valid. These observations motivate us to make relation semantics executable at runtime, allowing the system to derive the consequences of a change from both the relation type and the current execution state.

Figure 2 makes this distinction concrete. In an AsgardBench coffee task (Tupini et al., 2026), brewing is initially rejected because the mug is dirty. FlatReplan rearranges the remaining actions but leaves the violated precondition unresolved. GraSP (Xia et al., 2026) repairs the precondition and reaches drinking, but drinking itself changes the execution state: the mug becomes dirty again, leaving cleanup and storage work outstanding. Thus, even a successful action can establish one fact while invalidating another. Preserving completed work is therefore different from preserving stale state, and consistent state evolution is required throughout execution, not only when recovery is triggered.

Building on this insight, we introduce **ASTER** (Agent-State Transactional Execution Runtime), which makes state-transition semantics executable at runtime. ASTER represents committed state as a versioned typed hypergraph whose relations specify how changes propagate and where they stop. Given an event, the runtime derives the authorized scope of change and the conditions a valid successor should satisfy. The model proposes a concrete update within this scope, while ASTER checks its realized effects before atomic commit. In this way, the model retains flexibility over how to respond, while the runtime governs which state transitions become committed.

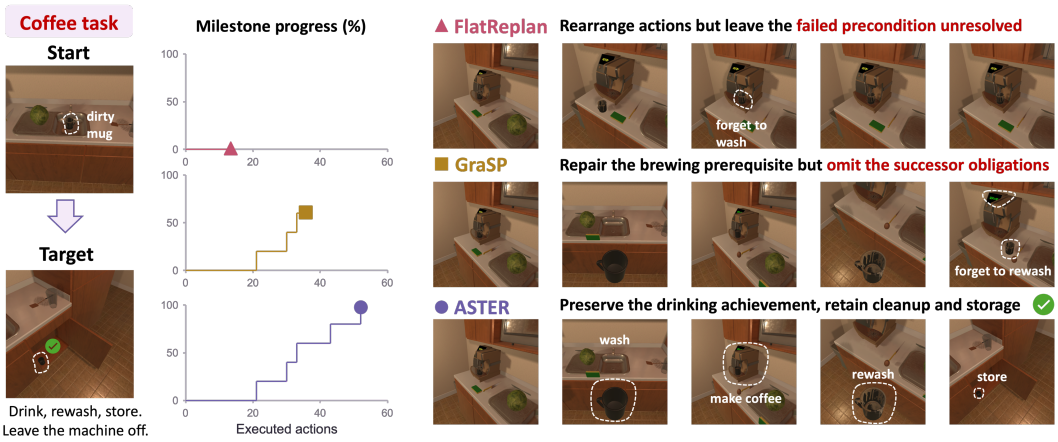


Figure 2: Illustrative example. In the AsgardBench coffee task (Tupini et al., 2026), ▲ FlatReplan globally replans remaining actions but leaves the violated brewing precondition unresolved. ■ GraSP repairs the precondition but drops cleanup obligations reintroduced by drinking. ● ASTER preserves completed work and carries these obligations into the successor state, completing all five milestones and satisfying all goal conditions.

We further analyze the conditions required for sound transition boundaries and certified commits. Our results characterize why typed joint relations provide information unavailable to untyped connectivity alone, and when ASTER's certified transitions satisfy transition integrity.

We evaluate ASTER on more than 1,200 tasks across nine benchmark settings spanning embodied execution, multi-agent coordination, interactive service and web tasks, and code generation. Across these settings, ASTER consistently outperforms the baselines, with gains of 3.0–23.6 percentage points over the stronger baseline in each setting (Figure 1). Controlled diagnostics probe whether event consequences are incorporated while unaffected state is preserved, and ablations isolate the contribution of executable transition semantics beyond structured state alone. The gains remain consistent across reasoning budgets and backbone configurations. Finally, when integrated into Codex (OpenAI, 2026a), ASTER further improves SWE-Amend success, indicating that runtime control over committed state can benefit existing agent systems beyond our primary ReAct setup.

Our contributions can be summarized as follows:

- We identify the *control gap* in persistent language-model agents, distinct from the context gap, and formalize valid state evolution through transition integrity.
- We introduce ASTER, a transactional runtime that makes typed state relations executable and certifies model-proposed successors before atomic commit.
- We provide theoretical analyses and evaluate ASTER across nine benchmark settings with controlled diagnostics, ablations, reasoning and backbone studies, and integration within an existing agent harness.

2 Transition Integrity under Runtime Change

Once an agent has accumulated committed execution state, a new event can invalidate some of it while leaving the rest valid. Let S_t denote the committed state when an event e_t arrives, and let S_{t+1} denote a candidate successor. Such events may arise from execution, observation, human input, peer activity, or environmental change. Giving the model access to this state does not by itself constrain the revision that becomes committed. This naturally leaves a question: what makes S_{t+1} a valid successor of S_t under e_t?

For a given (S_t, e_t), correctness has two sides. The event induces consequences that characterize a valid successor, captured by the ideal successor obligations $O_t^\star$, and an authority boundary over committed state, captured by the ideal writable region $W_t^\star$. The former describes what the successor reflects; the latter identifies which entities are legitimately open to change. Together, they characterize admissible revisions without fixing a particular repair.

Definition 2.1 (Transition integrity). *Let $\Delta_t = \Delta(S_t, S_{t+1})$ denote the entities affected by the realized edits, including changes to attributes and relation incidence. A transition satisfies transition integrity under e_t when*

$$\underbrace{S_{t+1} \models O_t^\star}_{\text{INCORPORATION}} \quad \wedge \quad \underbrace{\Delta_t \subseteq W_t^\star}_{\text{PRESERVATION}} . \tag{1}$$

Equation (1) explains why successor feasibility and edit size are insufficient on their own. A feasible successor may violate PRESERVATION by rewriting unaffected state, while a small edit may violate INCORPORATION by leaving an event consequence unresolved. The criterion applies to the transition from S_t to S_{t+1}; an event may leave the original goal unreachable even when both conditions hold.

At runtime, however, the ideal $W_t^\star$ and $O_t^\star$ cannot generally be recovered exactly from S_t and e_t. Their exact forms may depend on downstream consequences and execution conditions that are not yet observable at time t. Providing more context can help the model infer these semantics, but it does not make them conditions on what the runtime accepts: a violating revision may still be proposed, and a valid one may become stale before it is installed. Closing the control gap therefore requires moving transition semantics from context into the state-update path itself. Section 3 introduces ASTER as the runtime that realizes this idea.

3 ASTER: AGENT-STATE TRANSACTIONAL EXECUTION RUNTIME

ASTER moves state-transition control from model inference into the runtime. Grounding fixes what an event refers to, typed relation contracts derive which state may change and the conditions that characterize an admissible successor, and certification admits the resulting revision only against the version from which it was derived.

3.1 EXECUTABLE STATE REPRESENTATION

ASTER represents the executing state with explicit entities and relations, preserves that state across revisions, and equips its relations with semantics for runtime change.

Typed hypergraph. At logical time t, the structured execution state is a typed attributed hypergraph

$$\mathcal{G}_t = (V_t, E_t, \alpha_t), \tag{2}$$

where entities in V_t represent tasks, results, resources, observations, and commitments, and α_t stores their execution-dependent attributes. Each relation is a directed typed hyperedge

$$h = (T_h, H_h, \tau_h) \in E_t, \qquad \varnothing \neq T_h, H_h \subseteq V_t, \tag{3}$$

where T_h and H_h are its tail and head participants and τ_h denotes its relation type. Hyperedges retain joint conditions such as *all-of*, *any-of*, and k-of-n within a single relation. Losing one participant can therefore invalidate an *all-of* dependency while leaving an *any-of* dependency supported by another.

Versioned state. ASTER places this graph in a persistent committed state,

$$S_t = (\nu_t, \mathcal{G}_t, L_t), \tag{4}$$

where ν_t identifies the current version and L_t records the accepted events and revisions that produced it. Entity identities persist across versions, while properties such as lifecycle status, resource holdings, and observation freshness evolve in α_t. A task can thus progress from pending to running or completed without changing the object referenced by the rest of the state.

Executable relations. The hypergraph specifies which entities participate in a relation; its type specifies how that relation responds to change. Each relation type r is associated with a state-conditioned contract

$$\rho_r = \big(\text{activate}_r,\ \text{propagate}_r,\ \text{obligate}_r,\ \text{stop}_r\big), \tag{5}$$

which evaluates the joint condition, determines where a change propagates, derives conditions on the successor, and identifies where propagation stops. Because these operators are evaluated against the current execution state, the same structural relation can induce different consequences as its participants evolve.

ASTER provides five contract families: `Requires`, `Decomposes`, `Temporal`, `Contends`, and `Evidential`. A domain schema binds its relation vocabulary to these families, while the instance graph evolves through schema-conforming revisions. The binding remains fixed during a revision, providing the transition semantics used by the boundary derivation in Section 3.2. Full contract definitions are given in Table B.1 and 9.

3.2 Revision Boundary Derivation

A runtime event can affect more state than it explicitly names. ASTER derives the scope and conditions of revision before the model chooses a repair.

Event grounding. For an event e_t, ASTER takes a committed snapshot S_t and constructs a grounding g_t containing the interpreted change and its seed entities. Runtime identifiers can be resolved directly; free-form events are interpreted by the model. Retaining the change distinguishes cancellation from retry even when both concern the same task. The runtime validates the grounding against the committed version ν_t, the schema and protected anchors. These checks establish well-formedness; semantic interpretation remains a model responsibility.

Typed closure. The grounded change supplies direct successor obligations, and its seed entities initialize propagation. ASTER inspects incident hyperedges and evaluates their contracts on the unchanged snapshot S_t. A contract can extend the writable region, emit further obligations, or retain evidence that justifies stopping. Newly affected participants expose further relations, and evaluation continues until no new affected state or obligation is produced:

$$(\widetilde{\boldsymbol{W}}_t, \boldsymbol{B}_t, \widetilde{\boldsymbol{O}}_t) = \mathrm{Close}_\kappa(S_t, e_t, g_t), \tag{6}$$

where κ is the fixed contract vocabulary. Here $\widetilde{\boldsymbol{W}}_t$ is the derived writable region, and $\widetilde{\boldsymbol{O}}_t$ contains both direct and propagated successor conditions. Direct obligations are retained even when the event's targets have no downstream relations.

Propagation can stop where an alternative remains available or a completed result remains valid. After closure stabilizes, the retained witnesses form the read-only boundary evidence $\boldsymbol{B}_t$ only where they remain outside $\widetilde{\boldsymbol{W}}_t$; a participant reached through another active relation is writable instead. Thus $\boldsymbol{B}_t \cap \widetilde{\boldsymbol{W}}_t = \varnothing$. Each witness retains the relation and state attributes that justify the stopping decision. Appendix B.3 gives the convergence conditions and traversal analysis.

3.3 Bounded Revision and Certified Commit

With the revision boundary and successor obligations derived, ASTER turns them into an executable transaction. The model proposes a concrete update within the boundary, while the runtime checks its realized effects and commits the successor only after certification.

Bounded revision. The model now chooses a concrete patch π under the derived boundary. It receives the event, writable state, boundary witnesses with their typed relation context, and successor obligations. The patch uses schema-conforming attribute and graph edits and may revise any authorized subset of $\widetilde{\boldsymbol{W}}_t$: membership grants permission to change an entity without requiring that it change. Relevant observations and execution history can inform the proposal without enlarging write authority.

Successor certification. ASTER applies each candidate patch to a fresh shadow copy of the base snapshot, obtaining $\bar{S}_t = \mathrm{Shadow}(S_t, \pi)$. Certification checks this realized successor rather than relying on the patch's declared targets. It verifies that the base version and authorization remain valid,

$$\Delta(S_t, \bar{S}_t) \subseteq \widetilde{\boldsymbol{W}}_t, \qquad \bar{S}_t \models \widetilde{\boldsymbol{O}}_t, \tag{7}$$

and that the schema, graph, lifecycle, resource and registered domain invariants hold. The realized edit support includes changes to relation incidence, so naming only a writable target does not conceal effects on frozen participants. Structural predicates are checked directly; domain-dependent obligations use registered verifiers. The certificate records the discharged predicates and their supporting evidence, so the guarantee a commit carries is auditable rather than assumed.

Atomic commit. On successful certification, ASTER atomically checks the expected version and unused authorization, installs the certified successor, advances the version and records the transition.

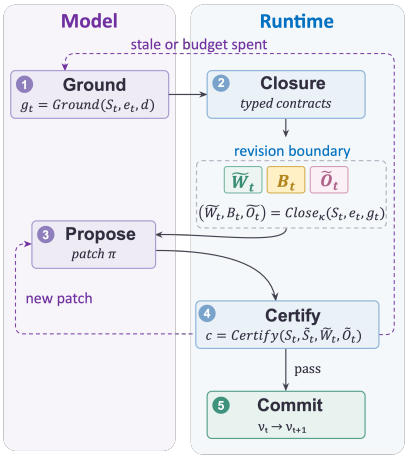


Figure 3: ASTER overview. The model grounds an event and proposes a patch; the runtime derives its boundary and obligations, certifies the successor, and commits atomically.

Table 1: Task success on each benchmark. Entries report success rates; bold marks the best results.

Benchmark	ASTER	GraSP	FlatReplan	Codex
Embodied task execution				
Single-agent execution				
ALFWorld	**70.15%**	58.21%	65.67%	61.19%
ScienceWorld	**49.31%**	37.50%	39.58%	43.75%
Visually grounded execution				
AsgardBench	**74.07%**	64.81%	62.96%[§]	—
Multi-agent coordination				
PARTNR	**85.00%**	76.00%	74.00%	26.00%
ParaCook	**79.17%**	55.56%	26.39%	43.06%
Interactive service and web tasks				
τ^2-bench	**86.8%**	82.3%	80.8%	70.3%
WebShop	**36.00%**	27.67%	27.33%	32.33%
AutomationBench[‡]	**26.67%**	18.33%	13.33%	16.67%
Code generation				
SWE-Amend[†]	**61.00%**	56.00%	58.00%	51.00%

Notes. [†] On SWE-Amend, ASTER, GraSP, and FlatReplan run as planning plugins within the Codex harness; Codex is unmodified. [§] Asgard-Base replaces FlatReplan on AsgardBench. [‡] AutomationBench uses sales-domain tasks.

Binding the final version check to installation prevents a candidate certified against one snapshot from overwriting a newer commit.

An invalid patch or failed check returns a diagnostic without installing the candidate. The model can revise its patch under the same boundary for up to K_p attempts. Exhausting this budget or encountering a version conflict restarts grounding and closure from a current snapshot, within the grounding budget K_g; Figure 3 shows these two retry levels. If no attempt commits within these budgets, the transaction returns `Deferred` with its last diagnostic. Algorithm 1 provides the pseudocode.

Commit guarantee. Every successful commit respects the derived write authority, satisfies the derived obligations, and preserves registered invariants. Appendix B.4 shows that these checks guarantee transition integrity when the derived authority lies within the ideal writable region and the derived obligations entail the ideal ones (Equation (13)). We further evaluate incorporation and preservation directly against oracle annotations in Appendix F.

4 Experiments

Our experiments aim to investigate the following questions: **(1)** How effective is ASTER across agent environments with different execution demands, and what computation accompanies its gains? **(2)** Which mechanisms of ASTER account for the observed improvements, and how do they affect agent behavior under runtime change? **(3)** Do these gains persist across agent harnesses, reasoning budgets, and backbone configurations?

4.1 Experimental Setup

Benchmarks. We evaluate ASTER on nine external agent benchmarks spanning five settings: single-agent embodied execution (ALFWorld, Shridhar et al., 2021; ScienceWorld, Wang et al., 2022), visually grounded embodied execution (AsgardBench, Tupini et al., 2026), multi-agent coordination (ParaCook, Zhang et al., 2026; PARTNR, Chang et al., 2025), interactive service and web tasks (τ^2-bench, Barres et al., 2025; WebShop, Yao et al., 2022; AutomationBench, Shepard & Salimans, 2026), and code generation (SWE-Amend, Jimenez et al., 2024). Each benchmark contributes a frozen matched task set shared by all compared methods. Unless otherwise stated, experiments use `gpt-5.4-mini` with reasoning effort `low`. Benchmark-specific task sets and protocols are detailed in Appendix E.

Baselines. We compare against FlatReplan, which maintains a flat ordered task list and replans after execution deviations, and GraSP (Xia et al., 2026), which maintains a typed dependency graph and performs bounded local repair. On AsgardBench, we additionally report Asgard-Base, the benchmark's released full-replanning scaffold, while Codex serves as a same-task reference cohort in its own harness. On SWE-Amend, ASTER, GraSP and FlatReplan are also plugged into Codex in place of its planning module. For the primary comparisons, ASTER, GraSP and FlatReplan share the same ReAct actor, prompts, tool and environment interfaces, backbone, decoding configuration and benchmark-specific budgets.

4.2 Evaluation Across Agent Environments

Across nine agent benchmarks with diverse execution demands, ASTER consistently achieves the highest task success, with gains of 3.00–23.61 percentage points over the stronger baseline (Table 1). Figure 4 summarizes the associated computation.

Embodied execution. ASTER improves embodied execution across single-agent, visually grounded, and multi-agent settings, covering ALFWorld, ScienceWorld, AsgardBench, ParaCook, and PARTNR. Figure 2 offers a representative case on AsgardBench: GraSP repairs the failed brewing precondition but misses the cleanup reintroduced by drinking, whereas ASTER preserves completed work while carrying the remaining requirements forward.

Multi-agent coordination. We next evaluate centralized coordination on ParaCook and distributed coordination on PARTNR, where peer actions continually change shared resources, dependencies, and commitments. These settings naturally expose the runtime events considered in our formulation: one agent may complete shared work, release a resource, or change what remains for another. ASTER shows its largest gain on ParaCook, reaching 79.17% success compared with 55.56% for GraSP. On PARTNR, ASTER reaches 85.00% task success on PARTNR against 76.00% for GraSP and 74.00% for FlatReplan. This gap suggests that coordination failures often arise not from lack of progress, but from losing track of the remaining dependencies and commitments needed to finish the task.

Interactive service and web tasks. **Interactive service and web tasks.** We further evaluate ASTER on τ^2-bench, WebShop, and AutomationBench. On τ^2-bench, where symptoms may have multiple concurrent causes, ASTER reaches 86.8% success while reducing human escalation to below 1% (Figure 4). The advantage extends to WebShop and AutomationBench, with gains of over eight points on both, covering product search and cross-application workflows.

Code generation. We further evaluate long-horizon code editing on SWE-Amend, where an additional requirement is introduced after the initial code changes (Appendix E). Integrated into the existing Codex harness, ASTER improves success from 51% to 61% and outperforms other baselines, highlighting its use as a reusable planning-state runtime beyond our React actor loop.

Computational cost. Figure 4 compares task success with model-call and input-token usage. ASTER uses fewer model calls than both planning baselines on seven of the eight benchmarks. Input tokens per call are also at or below the baselines on six of the eight benchmarks. Thus, the observed performance gains are not driven by a systematically larger inference budget; in most settings, ASTER achieves higher success with fewer model calls and comparable or lower per-call input usage.

4.3 Ablation Studies and Analysis

We further investigate which mechanisms contribute to ASTER and how its advantage changes with the reasoning capability of the backbone. Controlled diagnostics are provided in Appendix F.

Effects of each mechanism. We ablate four components on the ALFWorld subset (Figure 5). Removing successor obligations and write authority has the largest effect, reducing solved tasks from 28 to 22 and 24, respectively. Without successor obligations, ASTER is only one task above FlatReplan, suggesting that bounding where state may change is insufficient without also specifying what the successor must satisfy. The cost profile separates the variants more clearly. The full system incurs fewer precondition failures, uses fewer model calls, and reaches the call cap less often than every ablation. Interestingly, removing joint relations loses only two tasks, yet is the most expensive variant on three of the four cost measures.

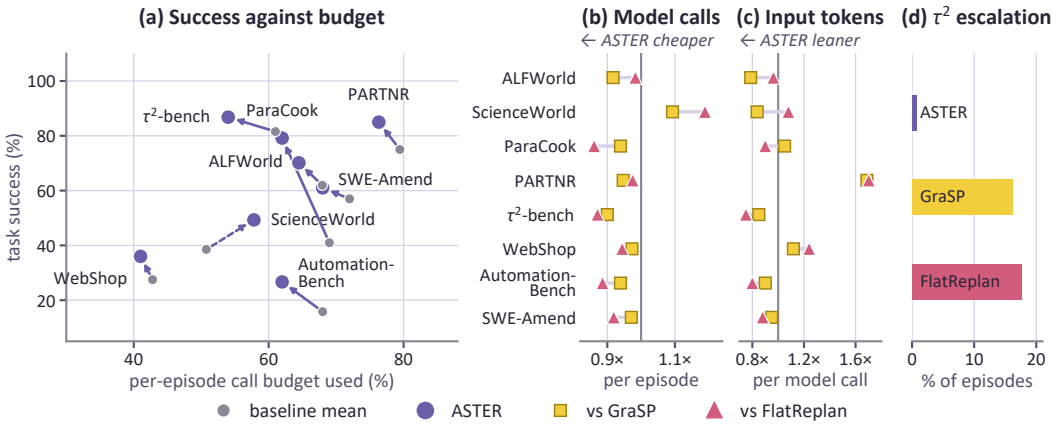
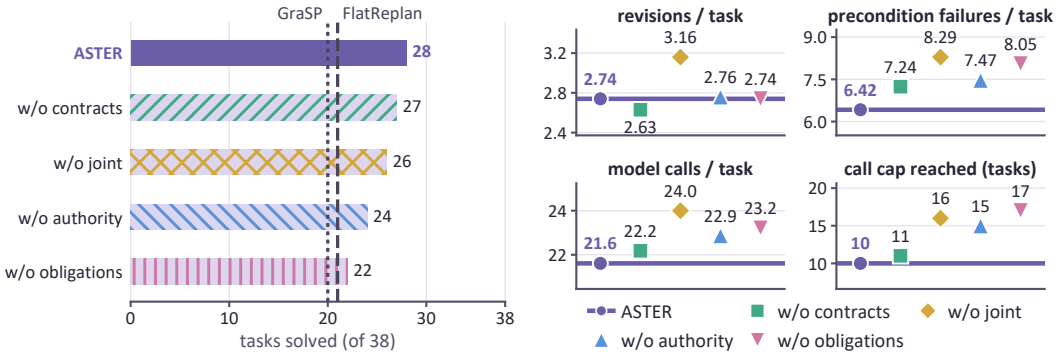


Figure 4: Behavioural profile. **(a)** Task success versus call-budget usage; arrows run from the mean of GraSP and FlatReplan to ASTER. **(b)** Model calls per episode and **(c)** input tokens per call, shown as ASTER-to-baseline ratios; values below 1 indicate lower usage. **(d)** Human escalation rates on τ^2-bench. Quantities are pooled per method over all episodes.



(a) Tasks solved **(b)** Cost profile

Figure 5: Effects of each mechanism. Each mechanism is removed individually on a 38-task ALFWorld subset, holding the actor, backbone, and budget fixed. **(a)** Tasks solved; dashed lines mark GraSP and FlatReplan. **(b)** Revisions, precondition failures, model calls, and budget exhaustion. The budget-exhaustion count includes tasks that succeed on the final allowed call.

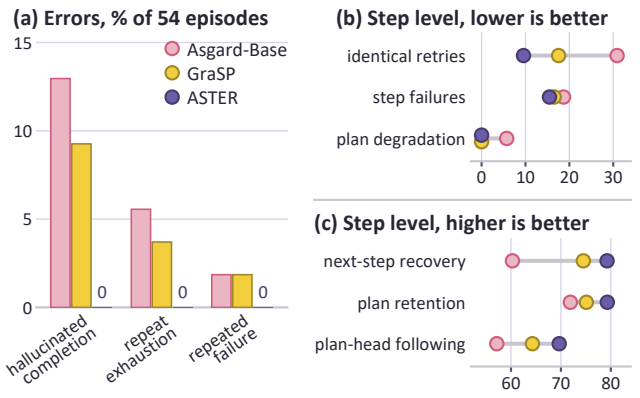


Figure 6: AsgardBench diagnostics. Results on 54 tasks excluding d-variants. (a) Episode-level error rates, all zero for ASTER. (b,c) Step-level rates pooled across tasks; lower/higher is better, respectively.

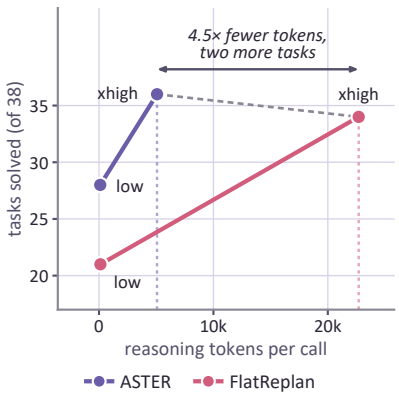


Figure 7: Reasoning budget. Tasks solved versus mean reasoning tokens per call over 38 ALFWorld tasks.

Behavioral diagnostics. We examine execution behavior on AsgardBench, which records the plan and termination cause at every step (Figure 6). ASTER exhibits no hallucinated completion, repeat exhaustion, or repeated-failure termination across 54 episodes; its failures arise only from budget exhaustion. Notably, hallucinated completion occurs in 13.0% of Asgard-Base episodes and under GraSP, despite GraSP exposing a typed plan graph to the model. ASTER also improves plan retention and plan-head following while reducing retries and step failures. These results suggest that structured plans can guide execution, but runtime control is needed for their state to govern it reliably.

Backbone and reasoning budget. We vary reasoning effort and backbone capability (Figures 7 and 8). On the same 38-task ALFWorld subset, moving from `low` to `xhigh` reduces the ASTER–FlatReplan gap from seven tasks to two; at `xhigh`, ASTER solves 36 tasks with 5.1k reasoning tokens per call, versus 34 tasks and 22.7k for FlatReplan. Across ParaCook configurations, ASTER leads in all unsaturated settings, while all methods reach 100% with `Qwen3.8-27B` thinking enabled. ASTER's advantage persists under stronger reasoning at a fraction of the reasoning cost.

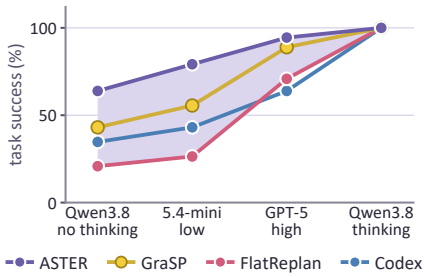


Figure 8: Backbone and reasoning configurations. ParaCook task success across four configurations.

5 Related Work

Persistent agent state and context management. Persistent agents retain experience through memories (Park et al., 2023), reusable skills (Wang et al., 2024; 2025), and task structures (Shen et al., 2023; Kim et al., 2024). Cognitive architectures organize these components into systems that maintain state across successive decisions (Sumers et al., 2024). Context management addresses which parts of that state are exposed to the model at a given execution frontier (Guo et al., 2026), while checkpointing supports recovery and continuation from stored execution records (LangChain, 2026b). Together, these approaches support the retention, retrieval and reuse of prior work, providing the state on which subsequent planning and execution depend.

LLM planning and plan repair. LLM agents interleave reasoning and action (Yao et al., 2023), produce structured plans or executable programs (Ahn et al., 2022; Singh et al., 2023; Liang et al., 2023; Liu et al., 2023a), and revise execution from feedback or human input (Huang et al., 2023; Ren et al., 2023; Liu et al., 2023b). Classical planning has long studied plan stability, adaptation, and continual planning under change (Fox et al., 2006; Gerevini & Serina, 2010; Brenner & Nebel, 2009; Cashmore et al., 2015; Percassi et al., 2023). Closest to our setting, Xia et al. (2026) bring structured local repair to LLM agents through typed plan graphs, motivating GraSP as our primary structured-repair baseline. ASTER addresses a complementary runtime problem: it derives event-conditioned revision boundaries and verifies that committed successors incorporate required changes while preserving unaffected state.

Additional related work is discussed in Appendix A.

6 Conclusion and Discussion

This paper identifies a control gap in language-model agents: structured state can inform model decisions without governing how committed state changes. We formalize transition integrity and introduce ASTER, which makes state-transition semantics executable through typed relations, successor certification, and atomic commit. Across nine agent benchmarks, ASTER consistently improves task success over replanning and structured-repair baselines, while its integration with Codex shows that the runtime is not tied to the React actor used in our main experiments.

Looking ahead, two directions merit further investigation. Executable relations could be extended to other forms of persistent state, such as memories and reusable skills, while certification feedback could be used to improve revision policies without removing independent runtime checks. More broadly, our work suggests that persistent agents benefit not only from richer state representations, but also from executable semantics that govern how those states evolve.

Reproducibility Statement

We provide the theoretical, implementation, and experimental details needed to reproduce our results. Section 4.1 describes the shared experimental setup. Appendix B states the assumptions and proofs, Appendix C specifies the runtime and transaction protocol, and Appendix F describes the controlled transition-integrity diagnostics. Appendix E documents the frozen task sets (Table E.2), run-selection rules, model configurations, call budgets, and benchmark-specific protocols; Appendix D provides the model-facing prompts.

AI Use Statement

We used generative AI tools in three parts of this work: drafting and editing the text of this paper, implementing code, and running and analysing experiments. We did not use generative AI tools for research ideation. The problem formulation, the transition-integrity framing, the contract and transaction design, and the claims made here are the authors' own.

AI-written code was read, executed, and tested by the authors before any result derived from it entered the paper; AI-assisted analyses were checked against the raw experiment logs; and AI-assisted text was checked against those results and against the cited literature. We take responsibility for the final content of this work, including all text, claims and artifacts produced with the aid of generative AI.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022. URL https://arxiv.org/abs/2204.01691.

Anthropic. Claude code agent sdk reference. https://code.claude.com/docs/en/agent-sdk/typescript, 2026. Accessed 2026-07-07.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025. URL https://arxiv.org/abs/2506.07982.

Michael Brenner and Bernhard Nebel. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, 2009. doi: 10.1007/s10458-009-9081-1. URL https://doi.org/10.1007/s10458-009-9081-1.

Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994. doi: 10.1016/0004-3702(94)90081-7. URL https://doi.org/10.1016/0004-3702(94)90081-7.

Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcis Palomeras, Natalia Hurtos, and Marc Carreras. ROSPlan: Planning in the robot operating system. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling*, pp. 333–341, 2015. doi: 10.1609/icaps.v25i1.13699. URL https://ojs.aaai.org/index.php/ICAPS/article/view/13699.

Matthew Chang, Gunjan Chhablani, Alexander Clegg, Mikael Dallaire Cote, Ruta Desai, Michal Hlavac, Vladimir Karashchuk, Jacob Krantz, Roozbeh Mottaghi, Priyam Parashar, Siddharth Patki, Ishita Prasad, Xavier Puig, Akshara Rai, Ram Ramrakhya, Daniel Tran, Joanne Truong, John M. Turner, Eric Undersander, and Tsung-Yen Yang. PARTNR: A benchmark for planning and reasoning in embodied multi-agent tasks. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a3cf318fbeec1126da21e9185ae9908c-Abstract-Conference.html.

Junting Chen, Checheng Yu, Xunzhe Zhou, Tianqi Xu, Yao Mu, Mengkang Hu, Wenqi Shao, Yikai Wang, Guohao Li, and Lin Shao. EMOS: Embodiment-aware heterogeneous multi-robot operating system with LLM agents. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Ey8KcabBpB.

Michele Colledanchise and Petter Ogren. *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018. doi: 10.1201/9780429489105.

Vardhan Dongre and Dilek Hakkani-Tür. Embodied multi-agent coordination by aligning world models through dialogue. In *Proceedings of the 27th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pp. 293–306. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.sigdial-1.21/.

Kutluhan Erol, James Hendler, and Dana S. Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18:69–93, 1996. doi: 10.1007/BF02136175. URL https://doi.org/10.1007/BF02136175.

Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling*, pp. 212–221, 2006. URL https://strathprints.strath.ac.uk/2776/.

Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2–3):177–201, 1993. doi: 10.1016/0166-218X(93)90045-P. URL https://doi.org/10.1016/0166-218X(93)90045-P.

Alfonso E. Gerevini and Ivan Serina. Efficient plan adaptation through replanning windows and heuristic goals. *Fundamenta Informaticae*, 102(3–4):287–323, 2010. doi: 10.3233/FI-2010-309. URL https://doi.org/10.3233/FI-2010-309.

Brian P. Gerkey and Maja J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9):939–954, 2004. doi: 10.1177/0278364904045564.

Shuhan Guo, Kun Zhang, Haifei Liu, Xingyu Gao, Yongqi Zhang, Yaqing Wang, and Quanming Yao. ContextFlow: Hierarchical task-state alignment for long-horizon embodied agents. *arXiv preprint arXiv:2605.19314*, 2026. URL https://arxiv.org/abs/2605.19314.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Tomas Jackson, Noah Brown, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models. In *Proceedings of the 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 1769–1782. PMLR, 2023. URL https://proceedings.mlr.press/v205/huang23c.html.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 24370–24391. PMLR, 2024. URL https://proceedings.mlr.press/v235/kim24y.html.

G. Ayorkor Korsah, Anthony Stentz, and M. Bernardine Dias. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, 32(12):1495–1512, 2013. doi: 10.1177/0278364913496484.

LangChain. Langchain prebuilt middleware: Todolistmiddleware. https://docs.langchain.com/oss/python/langchain/middleware/built-in, 2026a. Accessed 2026-07-07.

LangChain. Langgraph persistence. https://docs.langchain.com/oss/python/langgraph/persistence, 2026b. Accessed 2026-07-07.

Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. Agent-oriented planning in multi-agent systems. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=EqcLAU6gyU.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *IEEE International Conference on Robotics and Automation*, pp. 9493–9500, 2023. doi: 10.1109/ICRA48891.2023.10160591. URL https://doi.org/10.1109/ICRA48891.2023.10160591.

Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony G. Cohn, and Janet B. Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning. In *International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*. PMLR, 2024. URL https://proceedings.mlr.press/v235/lin24k.html.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023a. URL https://arxiv.org/abs/2304.11477.

Jie Liu, Pan Zhou, Yingjun Du, Ah-Hwee Tan, Cees G. M. Snoek, Jan-Jakob Sonke, and Efstratios Gavves. CaPo: Cooperative plan optimization for efficient embodied multi-agent cooperation. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=KRv9NubipP.

Zeyi Liu, Arpit Bahety, and Shuran Song. REFLECT: Summarizing robot experiences for failure explanation and correction. In *Proceedings of the 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pp. 3468–3484. PMLR, 2023b. URL https://proceedings.mlr.press/v229/liu23g.html.

Alberto Martelli and Ugo Montanari. Additive AND/OR graphs. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pp. 1–11, 1973. URL https://www.ijcai.org/Proceedings/73/Papers/001.pdf.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL: The planning domain definition language, 1998. URL https://www.isi.edu/results/publications/62624/pddl-the-planning-domain-definition-language/.

Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4): 541–580, 1989. doi: 10.1109/5.24143. URL https://doi.org/10.1109/5.24143.

Kazuma Obata, Tatsuya Aoki, Takato Horii, Tadahiro Taniguchi, and Takayuki Nagai. LiP-LLM: Integrating linear programming and dependency graph with large language models for multi-robot task planning. *IEEE Robotics and Automation Letters*, 10(2):1122–1129, 2025. doi: 10.1109/LRA.2024.3519868. URL https://arxiv.org/abs/2410.21040.

OpenAI. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, 2024. Accessed 2026.

OpenAI. Codex cli. https://github.com/openai/codex, 2026a. Accessed 2026-07-07.

OpenAI. Codex cli model instructions for plan updates. https://github.com/openai/codex/blob/main/codex-rs/models-manager/models.json, 2026b. Accessed 2026-07-07.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Francesco Percassi, Enrico Scala, and Mauro Vallati. Fixing plans for PDDL+ problems: Theoretical and practical implications. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling*, pp. 324–333, 2023. doi: 10.1609/icaps.v33i1.27210. URL https://ojs.aaai.org/index.php/ICAPS/article/view/27210.

Allen Z. Ren, Anushri Dixit, Alexandra Bodrova, Sumeet Singh, Stephen Tu, Noah Brown, Peng Xu, Leila Takayama, Fei Xia, Jake Varley, Zhenjia Xu, Dorsa Sadigh, Andy Zeng, and Anirudha Majumdar. Robots that ask for help: Uncertainty alignment for large language model planners. In *Proceedings of the 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pp. 661–682. PMLR, 2023. URL https://proceedings.mlr.press/v229/ren23a.html.

Sven Schneider, Johannes Dyck, and Holger Giese. Formal verification of invariants for attributed graph transformation systems based on nested attributed graph conditions. In *Graph Transformation*, volume 12150 of *Lecture Notes in Computer Science*, pp. 257–275. Springer, 2020. doi: 10.1007/978-3-030-51372-6_15. URL https://doi.org/10.1007/978-3-030-51372-6_15.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI tasks with ChatGPT and its friends in hugging face. In *Advances in Neural Information Processing Systems*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/77c33e6a367922d003ff102ffb92b658-Abstract-Conference.html.

Daniel Shepard and Robin Salimans. AutomationBench. *arXiv preprint arXiv:2604.18934*, 2026. URL https://arxiv.org/abs/2604.18934. Benchmark and data: https://github.com/zapier/AutomationBench.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *IEEE International Conference on Robotics and Automation*, pp. 11523–11530, 2023. doi: 10.1109/ICRA48891.2023.10161317. URL https://doi.org/10.1109/ICRA48891.2023.10161317.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=1i6ZCvflQJ.

Alejandro Torreño, Eva Onaindia, Antonín Komenda, and Michal Štolba. Cooperative multi-agent planning: A survey. *ACM Computing Surveys*, 50(6):84:1–84:32, 2017. doi: 10.1145/3128584.

Andrea Tupini, Lars Liden, Reuben Tan, Yu Wang, and Jianfeng Gao. AsgardBench: Evaluating visually grounded interactive planning under minimal feedback. *arXiv preprint arXiv:2603.15888*, 2026. URL https://arxiv.org/abs/2603.15888.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. ScienceWorld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.emnlp-main.775. URL https://aclanthology.org/2022.emnlp-main.775/.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*. PMLR, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Tianle Xia, Lingxiang Hu, Yiding Sun, Ming Xu, Lan Xu, Siying Wang, Wei Xu, and Jie Jiang. GraSP: Graph-structured skill compositions for LLM agents. *arXiv preprint arXiv:2604.17870*, 2026. URL https://arxiv.org/abs/2604.17870.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B. Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=EnXJfQqy0K.

Shiqi Zhang, Xinbei Ma, Zouying Cao, Zhuosheng Zhang, and Hai Zhao. Plan-over-graph: Towards parallelable LLM agent schedule. *arXiv preprint arXiv:2502.14563*, 2025. URL https://arxiv.org/abs/2502.14563.

Shiqi Zhang, Xinbei Ma, Yunqing Xu, Zouying Cao, Pengrui Lu, Haobo Yuan, Tiancheng Shen, Zhuosheng Zhang, Hai Zhao, and Ming-Hsuan Yang. ParaCook: On time-efficient planning for multi-agent systems. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 27683–27701. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1378. URL https://aclanthology.org/2026.findings-acl.1378/.

Appendix

A Extended Related Work **15**

B Theoretical Analyses **16**

B.1 Setup and Notation 16

B.2 Conditions on Scope Selection 18

B.3 Properties of the Representation and Its Closure 20

B.4 Soundness of Certified Commits 22

C Runtime Implementation **23**

C.1 Typed Hypergraph and Persistent State 23

C.2 Grounding and Runtime Propagation 24

C.3 Bounded Revision 25

C.4 Certification and Atomic Commit 25

D Prompts and Instructions **26**

D.1 Shared ReAct Execution Prompt 26

D.2 Baseline Planning Prompts 27

D.3 ASTER Runtime Prompts 27

E Comparison Experiment Details **29**

E.1 LLM configurations 29

E.2 ASTER and Baseline Implementations 29

E.3 SWE-Amend Construction and Evaluation 30

E.4 Task Cohorts 30

E.5 Recorded ParaCook Execution Trace 31

F Controlled Diagnostics of Transition Integrity **31**

F.1 Oracle-Controlled Diagnostic 32

F.2 Revision Boundary 33

F.3 Transactional Enforcement 34

F.4 Mechanism Stress Tests 35

G Notation **36**

A Extended Related Work

Persistent agent state and context management. Memory systems, skill libraries, and reusable workflows retain experience for later decisions (Park et al., 2023; Wang et al., 2024; 2025), while task structures and cognitive architectures organize its use in planning and action selection (Shen et al., 2023; Kim et al., 2024; Sumers et al., 2024). Context management and checkpointing support

state exposure and persistence (Guo et al., 2026; LangChain, 2026b), and agent frameworks provide planning tools and execution records (LangChain, 2026a; OpenAI, 2026a;b; Anthropic, 2026). ASTER complements these mechanisms by placing event-driven transitions of persistent planning state under runtime control.

LLM planning, acting, and plan repair. ReAct interleaves reasoning and interaction (Yao et al., 2023); structured action and program generation connect language to execution (Ahn et al., 2022; Singh et al., 2023; Liang et al., 2023; Liu et al., 2023a); and feedback, failure analysis, and human assistance support online correction (Huang et al., 2023; Ren et al., 2023; Liu et al., 2023b). Classical plan repair retains useful structure under changing goals or world states (Fox et al., 2006; Gerevini & Serina, 2010; Brenner & Nebel, 2009), including during robotic execution (Cashmore et al., 2015; Percassi et al., 2023). Xia et al. (2026) bring typed plan graphs and bounded local repair to LLM agents. Beyond constructing a replacement plan, ASTER certifies transition integrity: the successor reflects the event's consequences while preserving state outside its authorized scope (Definition 2.1).

Structured planning and reactive execution. PDDL and STRIPS specify action preconditions and effects (McDermott et al., 1998; Bylander, 1994); hierarchical task networks encode task decomposition (Erol et al., 1996); behavior trees organize reactive control (Colledanchise & Ogren, 2018); and Petri nets model joint enablement and resource consumption (Murata, 1989). These formalisms provide explicit semantics for actions, task structure, and execution. ASTER uses typed relation contracts and registered domain verifiers to constrain revisions of a partially executed structure, including how running actions, acquired results, and existing commitments are updated.

Hypergraphs, graph rewriting, and runtime verification. Directed hypergraphs represent multi-way relations (Martelli & Montanari, 1973; Gallo et al., 1993), while attributed hypergraph rewriting formalizes edits to such structures (Schneider et al., 2020). In ASTER, typed hyperedges carry propagation and stopping conditions for deriving event-specific revision boundaries and successor obligations (Table B.1). The runtime checks candidate successors in shadow state and applies version-guarded atomic commit. Proposition B.10 shows that every certified commit changes only authorized state, discharges the derived obligations and preserves registered invariants.

Multi-agent planning, allocation, and coordination. Multi-agent planning and task allocation distribute work under dependency, capability, and resource constraints (Gerkey & Matarić, 2004; Korsah et al., 2013; Torreño et al., 2017). LLM-based approaches generate task graphs and coordinate allocation and scheduling (Li et al., 2025; Lin et al., 2024; Zhang et al., 2025; Obata et al., 2025), while cooperative architectures incorporate communication, execution progress, and heterogeneous robot capabilities (Zhang et al., 2024; Liu et al., 2025; Chen et al., 2025). ParaCook evaluates parallel embodied plans (Zhang et al., 2026); PARTNR documents collaboration failures such as repeating or undoing a partner's completed work (Chang et al., 2025), and further work examines communication and action conflicts (Dongre & Hakkani-Tür, 2026). ASTER complements allocation and communication by governing changes to committed planning state: peer updates enter as events, and the resulting revisions are checked for event consequences and preservation of unaffected commitments before commit.

B Theoretical Analyses

Here we give the analyses the body states in outline. We proceed in three steps. We first ask what a revision must *read* before it can bound itself, and answer with two impossibility results: a method that reads no more than it may write cannot detect under-propagation (Proposition B.4), and no untyped rule repairs this however much it reads (Proposition B.5). We then ask what the typed hypergraph *supplies* in their place, and show that ordinary graphs embed into it exactly (Proposition B.6), that the reverse encoding loses the operator deciding where propagation stops (Proposition B.7), and that the closure terminates (Proposition B.8) at a cost in inspected incidence (Proposition B.9). We close by asking what a certificate *guarantees*, and discharge both clauses of Equation (1) relative to the boundary the first two steps justify (Proposition B.10 and Corollary B.11).

B.1 Setup and Notation

Entities and contracts. Relations are addressed exactly as nodes are. The plan $\mathcal{G}_t$ of Equation (2) therefore has entity universe $U_t = V_t \sqcup E_t$, and every scope and closure below is defined over it. The

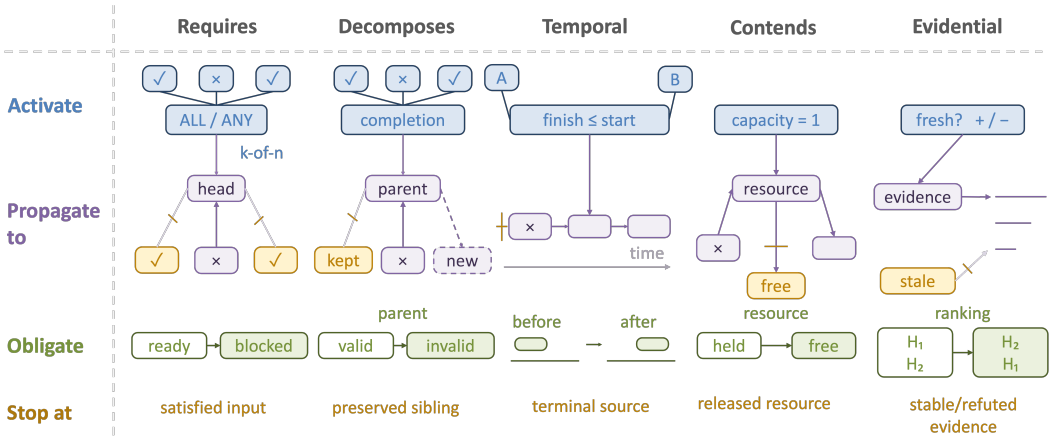


Figure 9: The five relation contracts on small examples. Rows follow the components of Equation (8). *Activate*: the condition under which the relation holds, over its participants. *Propagate to*: a change (×) travels along the arrows to the participants the contract names; gold entities are where stop_r halts it, marked by a bar across the edge. *Obligate*: the successor condition that arrival emits, drawn as the state before and after. *Stop at*: the halting predicate. Table B.1 gives the complete entries.

contract that Section 3.1 attributes to a relation type r, Equation (5), is restated here so that the proofs below can refer to its four operators by name:

$$\rho_r = \big(\mathrm{activate}_r,\ \mathrm{propagate}_r,\ \mathrm{obligate}_r,\ \mathrm{stop}_r\big). \tag{8}$$

Here $\mathrm{activate}_r$ is a predicate on the incident values that make the relation hold, $\mathrm{propagate}_r$ names the participants to which an affected one carries the event, $\mathrm{obligate}_r$ names the successor conditions that arrival emits, and stop_r is the predicate at which propagation halts. Table B.1 states the five contracts in full, and the proofs below use those identifiers; Figure 9 draws each contract on a small example.

Table B.1: Relation contracts ρ_r. One column per component of Equation (8). Two types with the same four entries are one type with an attribute, which is the filter that produces exactly five.

Type	Activate	Propagate to	Obligate	Stop at
REQUIRES	`all`, `any`, k-of-n	the head	block, re-enable	independently satisfied participant
DECOMPOSES	aggregate completion	the parent aggregate	invalidate, re-decompose	preserved sibling
TEMPORAL	precedence, window, deadline	incident actions	reschedule	*terminal* source
CONTENDS	capacity, mutual exclusion	*every* competitor	preempt, serialize, rebind	released resource
EVIDENTIAL	belief rule, polarity, freshness	hypothesis frontier	re-rank, keep provenance	refuted or stale evidence

Reading, writing and the untyped projection. Both results of Appendix B.2 quantify over what a method reads, so we first give reading a definition. For $X \subseteq U_t$ we write $S|_X$ for the *restriction* of a state to X, $\mathrm{Atoms}(X)$ for the edit atoms it touches, and $u(S)$ for the *untyped projection*:

$$S|_X = \big(X,\ \alpha_t|_X,\ \{h \in E_t : \iota(h) \subseteq X\}\big), \tag{9}$$

$$\mathrm{Atoms}(X) = \{\delta \in \mathfrak{A}_t : \mathrm{supp}(\delta) \cap X \neq \varnothing\}, \tag{10}$$

$$u(S) = \big(U_t,\ \alpha_t,\ \{\iota(h)\}_{h \in E_t}\big), \qquad \iota(h) = \{h\} \cup T_h \cup H_h. \tag{11}$$

Here supp returns the entities a single edit atom changes: changing an attribute supports its owner, and adding or removing an incidence supports both the relation and the participant at that incidence; on a set of atoms, supp is the union over its members. We write $D(S, S') \subseteq \mathfrak{A}_t$ for the atoms by

which two states differ, so the entity set of Equation (1) is $\Delta_t = \operatorname{supp} D(S_t, S_{t+1})$. Authority is checked on this realized support. A patch therefore cannot bypass preservation by declaring only the node it intends to delete.

Three consequences of Equations (9)–(11) are used repeatedly below. A relation with one participant outside X does not occur in $S|_X$ at all. An incidence atom joining a writable entity to a frozen one lies in $\mathrm{Atoms}(F_t)$. And u keeps attachment while discarding the types τ_h, hence the contracts of Equation (8). Inside a single construction we drop the time subscript and write $P^\star$ and $Q^\star$ for the reference patches of the two episodes being compared. In the constructions below the oracle authorizes exactly what its repair edits, $W_t^\star = \operatorname{supp} P^\star$, and every reference atom is required: a committed patch that misses one fails incorporation, and one that writes outside the support fails preservation.

Definition B.1 (Evidence and authorization)**.** *Fix (S_t, e_t) and the entity universe U_t. The evidence scope $C_t \subseteq U_t$ is what a method reads, and the writable scope $\widetilde{W}_t \subseteq U_t$ is what its patch may write. The frozen set is the complement $F_t = U_t \setminus \widetilde{W}_t$, and the boundary witnesses are $B_t = C_t \cap F_t$: the entities a method must see in order to know where to stop, and must not touch.*

For ASTER the two scopes are the ones Equation (6) returns. Propagation reaches $\widetilde{W}_t$, halts at B_t and reads nothing else, so $C_t = \widetilde{W}_t \cup B_t$ and the identity $B_t = C_t \cap F_t$ holds. A method that reads no more than it may write is the case $C_t = \widetilde{W}_t$, which is the hypothesis of Proposition B.4.

The propagation recurrence. Grounding returns a seed $Z^{(0)}$, and typed propagation closes it under the contracts of the relations it reaches. For a relation instance h of type r, the contract supplies a guarded transfer operator ϕ_h built from $\mathrm{propagate}_r$ and stop_r. This operator may require a joint match over T_h and may halt at its stopping predicate, so it is not ordinary adjacency. Its output stays within the relation, $\phi_h(Z) \subseteq \iota(h)$, since a contract transmits only to participants of the relation that carries it. At each step we apply the operators of the relations reached so far,

$$Z^{(k+1)} = Z^{(k)} \cup \bigcup_{h:\, \iota(h) \cap Z^{(k)} \neq \varnothing} \phi_h\big(Z^{(k)}\big), \qquad \mathrm{Closure}_\rho\big(\mathcal{G}_t, Z^{(0)}\big) = Z^{(\infty)}. \tag{12}$$

The guard is on incidence and not on adjacency. A relation is therefore visited once however many participants it joins, which is the property Proposition B.7 prices against a pairwise encoding.

B.2 Conditions on Scope Selection

Definition 2.1 constrains where a revision writes and leaves open what it writes there. The two results here are conditions on the region, not on the repair. Both are stated for an arbitrary scope-selection method, so we first record what such a method is assumed to be.

Assumption B.2 (Evidence determinacy)**.** The patch a scope-selection method commits is a function $\widehat{P}_t = \mathcal{M}(e_t, S|_{C_t})$ of the event and of the evidence projection alone.

Assumption B.3 (Untyped evidence)**.** A method satisfying Assumption B.2 is *untyped* if it factors through the untyped projection, that is, if $\mathcal{M}(e_t, S|_{C_t}) = \mathcal{M}'\big(e_t, u(S|_{C_t})\big)$.

Assumption B.2 holds for any deterministic repair procedure by construction: what it never reads cannot influence what it writes, and a randomized one is covered by conditioning on its draws. Reachability closures, k-hop neighbourhoods, rooted-subtree grants and list-position heuristics satisfy Assumption B.3 as well, since each computes its scope from attachment alone. Both proofs below follow the same plan. We construct a pair of episodes the method cannot tell apart, and we show their reference patches differ.

Proposition B.4 (Boundary evidence is necessary)**.** *Let $\widetilde{W}_t \subsetneq U_t$ and $C_t = \widetilde{W}_t$. Then $B_t = \varnothing$, and no method satisfying Assumption B.2 satisfies Equation (1) on every episode. There are episodes on which such a method under-propagates and on which the certificate of Section 3.3 nevertheless accepts.*

Proof. (i) Definition B.1 makes F_t the complement of $\widetilde{W}_t$, so $C_t = \widetilde{W}_t$ leaves no frozen entity in the evidence:

$$B_t = C_t \cap F_t = \widetilde{W}_t \cap (U_t \setminus \widetilde{W}_t) = \varnothing.$$

(ii) Next we construct the pair. Fix $p, a \in \widetilde{W}_t$ and $b, b' \in F_t$, let the event e report that p has failed, and let two episodes over the same universe carry the relations

$$\begin{aligned} S: &\quad T_h = \{p\},\ H_h = \{a\}, \quad T_{h'} = \{b'\},\ H_{h'} = \{b\}; \\ S': &\quad T_h = \{p\},\ H_h = \{a\}, \quad T_{h'} = \{a\},\ H_{h'} = \{b\}, \end{aligned}$$

with the relation entities placed so that $h \in \widetilde{W}_t$ and $h' \in F_t$. The episodes differ only in the tail of h': entirely frozen in S, crossing the boundary in S'. In both, $\iota(h') \not\subseteq \widetilde{W}_t$, so Equation (9) drops h' and the two restrictions coincide. The episodes are therefore indistinguishable to the method, while their reference patches differ:

$$S|_{\widetilde{W}_t} = S'|_{\widetilde{W}_t}, \qquad P^\star \subseteq \mathfrak{A}_t \setminus \mathrm{Atoms}(F_t), \qquad Q^\star \cap \mathrm{Atoms}(\{b\}) \neq \varnothing.$$

(iii) We now compare the committed patch against the two references. By Assumption B.2 with $C_t = \widetilde{W}_t$, and because atoms outside the grant are inexpressible,

$$\widehat{P}_t = \mathcal{M}(e, S|_{\widetilde{W}_t}) = \mathcal{M}(e, S'|_{\widetilde{W}_t}), \qquad \mathrm{supp}(\widehat{P}_t) \subseteq \mathfrak{A}_t \setminus \mathrm{Atoms}(F_t).$$

Combining this with (ii), we obtain

$$Q^\star \setminus \widehat{P}_t \ \supseteq\ Q^\star \cap \mathrm{Atoms}(\{b\}) \neq \varnothing.$$

On S' the method under-propagates, and the stale action at b remains executable.

(iv) Finally we check that the certificate does not separate the episodes. Its authority and preservation tier tests

$$\mathrm{supp}(\widehat{P}_t) \subseteq \mathfrak{A}_t \setminus \mathrm{Atoms}(F_t) \qquad \text{and} \qquad D(S_t, \bar{S}_t) \cap \mathrm{Atoms}(F_t) = \varnothing,$$

both of which are functions of the patch and of the grant, and both of which hold on S and S' alike. The clause that fails on S' is incorporation, which quantifies over $O_t^\star$ and hence over an entity the method never read. The only witness separating the two episodes is the pair (b, h'): an entity of F_t together with a relation crossing the boundary. That pair is B_t and the typed incidence attached to it, which is the claim. □

The hypothesis $\widetilde{W}_t \subsetneq U_t$ is what the proposition is about. A method may satisfy $C_t = \widetilde{W}_t$ and never under-propagate by taking $\widetilde{W}_t = U_t$. Then $F_t = \varnothing$, step (ii) has nowhere to place b, and B_t is empty without cost. That method is the full rebuild, which satisfies one clause of Equation (1) by maximally violating the other. Reading beyond $\widetilde{W}_t$ is therefore necessary for any method that draws a boundary at all. The next proposition shows it is not sufficient.

Proposition B.5 (Untyped evidence is insufficient). *Let a method satisfy Assumptions B.2 and B.3. Then* (i) *for every evidence scope C_t, including $C_t = U_t$, there are episodes on which it violates Equation (1); and* (ii) *if on some episode whose event reports the failure of a seed entity it grants a strict subset of the entities its evidence reaches, there are episodes on which it under-propagates.*

Proof. (i) We construct the pair first. Take entities p_1, p_2, a, c and two relations

$$T_h = \{p_1, p_2\}, \quad H_h = \{a\}, \qquad T_{h_c} = \{a\}, \quad H_{h_c} = \{c\},$$

and let the event e report that p_1 has failed. The two episodes differ only in the activation operator carried by h:

$$\begin{aligned} \mathrm{activate}_{\tau_h^S} = \texttt{all} &\implies \mathrm{Atoms}(\{a, c\}) \cap P^\star \neq \varnothing, \\ \mathrm{activate}_{\tau_h^{S'}} = \texttt{any} &\implies \mathrm{Atoms}(\{a, c\}) \cap Q^\star = \varnothing. \end{aligned}$$

Under `all` the loss of p_1 deactivates the requirement, the induced transition invalidates a, and transmission continues along h_c to c. Under `any` the entity a rebinds to p_2, and stop_r fires at the independently satisfied participant.

The two episodes have the same entities, node attributes, participant sets and event. By Equation (11) we therefore have $u(S) = u(S')$, and Assumption B.3 propagates this equality through the method:

$$u(S) = u(S') \implies u(S|_{C_t}) = u(S'|_{C_t}) \implies \mathcal{M}(e, S|_{C_t}) = \mathcal{M}(e, S'|_{C_t}) =: \widehat{P}_t$$

for every $C_t \subseteq U_t$. One patch now faces two references. Splitting the error into $P^\star \setminus \widehat{P}_t$, which is under-propagation, and $\widehat{P}_t \setminus P^\star$, which is over-propagation, and testing $\widehat{P}_t$ on the one set where the two references disagree, we obtain

$$\widehat{P}_t \cap \mathrm{Atoms}(\{a, c\}) = \varnothing \implies P^\star \setminus \widehat{P}_t \neq \varnothing \quad \text{on } S,$$
$$\widehat{P}_t \cap \mathrm{Atoms}(\{a, c\}) \neq \varnothing \implies \widehat{P}_t \setminus Q^\star \neq \varnothing \quad \text{on } S'.$$

One of the two errors is therefore incurred whichever way $\widehat{P}_t$ falls. Taking $C_t = U_t$ shows that enlarging the evidence does not break the tie: both episodes are already fully visible, and what distinguishes them is the type τ_h of a relation the method already sees, which u discards. The same construction runs with TEMPORAL against REQUIRES over one incidence, an ordering relation being discharged by a *terminal* source however it ended and a requirement only by a successful one.

(ii) Suppose that on some episode S whose event reports the failure of a seed entity, the method grants $\widehat{W}_t$ strictly inside the entities its evidence reaches. Then some reached entity x lies outside the grant, joined to the seed by a chain of relations inside C_t. Every quantity the grant may depend on is a function $\psi \circ u$ of the untyped projection, among them hop distance, subtree membership and position in a list. We now retype the chain and change nothing else: every relation joining the seed to x becomes REQUIRES with reading `all`, its tails on the seed side. Writing S' for the result, this surgery alters no entity, no attribute and no incidence set $\iota(h)$, only types and orientations, both of which Equation (11) discards. Hence

$$u(S) = u(S') \implies \psi\big(u(S)\big) = \psi\big(u(S')\big) \implies \widehat{W}_t(S) = \widehat{W}_t(S'),$$

and the grant on S' still excludes x. On S', however, the reported failure deactivates each `all` requirement in turn, and the reference patch carries the invalidation along the chain, so $Q^\star \cap \mathrm{Atoms}(\{x\}) \neq \varnothing$ while $\widehat{P}_t \cap \mathrm{Atoms}(\{x\}) = \varnothing$, atoms outside the grant being inexpressible. We obtain $Q^\star \setminus \widehat{P}_t \neq \varnothing$, which is under-propagation. No function of the untyped projection implements stop_r, and the proof is complete. □

Part (ii) leaves one alternative open: transmit across every relation reached, and grant the whole connected region. That avoids under-propagation by abandoning preservation. It is untyped reachability, and the untyped-reach arm of Appendix F (Table F.2) measures what it costs.

B.3 Properties of the Representation and Its Closure

The two propositions above say what a boundary must read. We now turn to what the typed hypergraph supplies. We begin with the direction that costs nothing.

Proposition B.6 (Ordinary-graph embedding)**.** *Every finite typed attributed directed graph G admits an incidence-, type- and attribute-preserving embedding into a hypergraph of the form of Equation (3) whose hyperedges all have one tail and one head.*

Proof. Let $G = (V, A, \tau_V^G, \tau_A^G, \alpha^G)$ be a typed attributed directed graph, its attributes carrying whatever temporal fragment it declares. We construct $\mathcal{G}_t$ by

$$V_t = V, \qquad E_t = \{h_a : a \in A\}, \qquad T_{h_a} = \{u\}, \quad H_{h_a} = \{v\} \text{ for } a = (u, v),$$

with $\tau_{h_a} = \tau_A^G(a)$ and $\alpha_t = \alpha^G$, and with the role τ_V^G assigns a node recorded among that node's attributes. Each arc becomes its own relation instance: instances are entities of U_t and carry their incidence and type as data, so distinct arcs, parallel ones included, yield distinct hyperedges h_a. Define $\Phi : V \sqcup A \to V_t \sqcup E_t$ by $\Phi|_V = \mathrm{id}_V$ and $\Phi(a) = h_a$. Then Φ is a bijection, and for every $a = (u, v) \in A$ we have

$$\Phi(u) \in T_{\Phi(a)}, \qquad \Phi(v) \in H_{\Phi(a)}, \qquad \tau_{\Phi(a)} = \tau_A^G(a),$$

with the analogous identities for Φ^{-1}. Hence Φ preserves direction, incidence, type and attributes in both directions, and G is isomorphic to its image. That image is exactly $\{h \in E_t : |T_h| = |H_h| = 1\}$, which is the claim. □

Task trees and skill DAGs are the cases $\tau_A^G \equiv$ DECOMPOSES and $\tau_A^G \equiv$ REQUIRES. This proposition therefore says that they are special cases of the representation and not approximations of it. The reverse direction fails, and it fails in the operator that decides where propagation stops.

Proposition B.7 (Pairwise encoding loses the activation operator)**.** *Let g send a relation to its pairwise encoding. Then* (i) *no function of $g(h)$ recovers which reading of* $\mathrm{activate}_r$ *the relation h carries, and* (ii) *the encoding of a symmetric* CONTENDS *relation over n competitors is quadratic in n, where the closure of Equation* (12) *visits one relation.*

Proof. (i) For a directed relation the pairwise encoding is $g(h) = T_h \times H_h$, which depends on h only through its incidence. Let ρ and ρ' be two contracts differing only in their activation operator. Then

$$g(h)\big|_\rho = T_h \times H_h = g(h)\big|_{\rho'}, \qquad \text{hence} \qquad \psi\big(g(h)\big)\big|_\rho = \psi\big(g(h)\big)\big|_{\rho'}$$

for every function ψ. The two contracts nonetheless differ on the event that one tail is lost: under `all` the relation deactivates, and under `any` it does not. Any ψ is therefore constant on a fibre on which the truth to be recovered is not. This is Proposition B.5 localized to a single relation.

(ii) Mutual exclusion over a competitor set K with $|K| = n$ is symmetric and relates every pair, so

$$g(h) = \big\{\{v, w\} : v \neq w \in K\big\}, \qquad |g(h)| = \binom{n}{2} = \Theta(n^2), \qquad \text{whereas} \qquad |\iota(h)| = n + 1.$$

Equation (12) guards on $\iota(h) \cap Z \neq \varnothing$ and so visits the single relation once, which completes the proof. □

Taken together, the two propositions say that the representation is a strict extension: it loses nothing an ordinary graph carries, and it carries an operator no pairwise encoding recovers. We turn next to what evaluating it costs.

Proposition B.8 (Finite selective closure)**.** *If $\mathcal{G}_t$ is finite and each ϕ_h is monotone, the iteration of Equation* (12) *reaches the least fixed point above $Z^{(0)}$ after at most $|U_t|$ strict expansions, and reaches the same fixed point under every fair evaluation order.*

Proof. Write one step of Equation (12) as

$$F(Z) = Z \cup \bigcup_{h:\, \iota(h) \cap Z \neq \varnothing} \phi_h(Z), \qquad Z^{(k+1)} = F\big(Z^{(k)}\big), \qquad Z^{(k)} \subseteq U_t,$$

the universe being finite and containing the hyperedges. First, F is inflationary through the union with Z. Second, F is monotone, because each ϕ_h is monotone by hypothesis and the guard is itself monotone:

$$Z \subseteq Z' \implies \{h : \iota(h) \cap Z \neq \varnothing\} \subseteq \{h : \iota(h) \cap Z' \neq \varnothing\} \implies F(Z) \subseteq F(Z').$$

The iterates therefore ascend inside U_t, and each strict step consumes at least one of its elements:

$$Z^{(k)} \subsetneq Z^{(k+1)} \implies \big|Z^{(k+1)}\big| \geq \big|Z^{(k)}\big| + 1.$$

At most $|U_t| - |Z^{(0)}| \leq |U_t|$ steps are thus strict. Writing $k^\star$ for the index after the last of them, we obtain $\mathrm{Closure}_\rho(\mathcal{G}_t, Z^{(0)}) = Z^{(k^\star)} = F(Z^{(k^\star)})$.

It remains to show this fixed point is least above the seed. Let Y satisfy $Y = F(Y)$ and $Z^{(0)} \subseteq Y$. Then

$$Z^{(0)} \subseteq Y, \qquad Z^{(k)} \subseteq Y \implies Z^{(k+1)} = F\big(Z^{(k)}\big) \subseteq F(Y) = Y,$$

so $Z^{(k^\star)} \subseteq Y$ by induction on k, and $Z^{(k^\star)}$ is the least fixed point above the seed.

Finally we release the evaluation order. Call an order *fair* if it eventually fires each relation whose guard holds. A single firing preserves containment in $Z^{(k^\star)}$: if $W \subseteq Z^{(k^\star)}$ and $\iota(h) \cap W \neq \varnothing$, the guard holds at $Z^{(k^\star)}$ a fortiori, so

$$W \cup \phi_h(W) \;\subseteq\; Z^{(k^\star)} \cup \phi_h\big(Z^{(k^\star)}\big) \;\subseteq\; F\big(Z^{(k^\star)}\big) = Z^{(k^\star)}.$$

The universe is finite, so a fair run stabilizes, and at its limit every guarded operator has fired without effect; the limit is therefore a fixed point above $Z^{(0)}$. Leastness bounds it from below by $Z^{(k^\star)}$, and the containment just shown bounds it from above, so every fair order reaches $Z^{(k^\star)}$, and the proof is complete. □

This proposition implies that the boundary the runtime returns does not depend on the order in which relations are visited. The next one bounds the work required to reach it.

Proposition B.9 (Traversal cost)**.** *Adopt the hypotheses of Proposition B.8, and let a worklist evaluation of Equation (12) satisfy* (a) *it notifies each incident relation once per newly added entity,* (b) *it maintains per relation a count of its participants already in Z, and* (c) *it emits from a firing only the members of $\phi_h(Z)$ not already in Z. Then it computes* $\mathrm{Closure}_\rho(\mathcal{G}_t, Z^{(0)})$ *in*

$$O\Big(\textstyle\sum_{h\in E_{\mathrm{visit}}}\big(|T_h|+|H_h|\big)\Big)$$

operations, where E_{visit} is the set of relations reached.

Proof. Split the work into the two things the evaluation does, namely notification and emission:

$$\mathrm{cost} = \underbrace{\textstyle\sum_{h\in E_{\mathrm{visit}}} n_h \cdot c_{\mathrm{notify}}}_{N} + \underbrace{\textstyle\sum_{h\in E_{\mathrm{visit}}} m_h \cdot c_{\mathrm{emit}}}_{E},$$

where n_h is the number of notifications h receives and m_h the number of entities it emits. We bound the four quantities in turn.

For n_h: the chain of Proposition B.8 is ascending, so an entity enters Z at most once and notifies each incident relation at most once, which gives $n_h \le |\iota(h)|$. For c_{notify}: monotonicity makes the count of (b) non-decreasing, so an activation predicate is re-evaluated by incrementing that count rather than by rescanning $\iota(h)$, and $c_{\mathrm{notify}} = O(1)$. For m_h: every firing satisfies $\phi_h(Z) \subseteq \iota(h)$ by the construction of ϕ_h (Appendix B.1), and by (c) an entity is emitted by h at most once, so $m_h \le |\iota(h)|$ however many times h fires. For c_{emit}: emitting one entity is $O(1)$.

Combining the four bounds, we obtain

$$\mathrm{cost} = O\Big(\sum_{h\in E_{\mathrm{visit}}} |\iota(h)|\Big) = O\Big(\sum_{h\in E_{\mathrm{visit}}} \big(1+|T_h|+|H_h|\big)\Big) = O\Big(\sum_{h\in E_{\mathrm{visit}}} \big(|T_h|+|H_h|\big)\Big),$$

the last step because T_h and H_h are non-empty by Equation (3), so that $1+|T_h|+|H_h| \le 2(|T_h|+|H_h|)$. No relation outside E_{visit} is inspected, so the bound is independent of $|U_t|$, which is the claim. □

Every contract in Table B.1 is monotone in the sense the two propositions require: a relation that has transmitted does not un-transmit when a further participant becomes affected. Monotonicity is a property of the accumulated consequences and not of the intermediate classification. A participant that provisionally witnesses a stop may therefore still be reached later by another relation, and the fixed point is what settles its class. Removing the writable participants from the retained evidence then makes $\boldsymbol{B}_t$ and $\widetilde{\boldsymbol{W}}_t$ disjoint by construction.

B.4 Soundness of Certified Commits

The results above concern the boundary. We close with what the transaction adds once a boundary is granted.

Proposition B.10 (Soundness of certified commits)**.** *Suppose certification checks the realized semantic edit support and successor predicates against one immutable base version, and suppose atomic commit verifies that this version is current and its authorization unused. Then every successful commit changes only authorized state, satisfies the derived obligations, and preserves all registered invariants.*

Proof. The authority check bounds every realized edit by the derived authority, and the obligation and invariant checks establish the corresponding predicates on the candidate. Atomic installation succeeds only while the checked base version and authorization remain valid. The installed successor is therefore the candidate checked against the state it actually replaces, and an unsuccessful attempt installs no candidate, which is the claim. The statement is about represented state: an external action enters that state through the outcome its registered protocol reports, and is checked there. □

Corollary B.11 (Transition integrity of certified commits)**.** *If, for the event and state under consideration,*

$$\widetilde{W}_t \subseteq W_t^\star \qquad \textit{and} \qquad \widetilde{O}_t \models O_t^\star, \tag{13}$$

then a certified commit satisfies Equation (1)*.*

Proof. Authority gives $\Delta_t \subseteq \widetilde{W}_t \subseteq W_t^\star$, which is preservation. The obligation check gives $S_{t+1} \models \widetilde{O}_t$, and entailment yields $S_{t+1} \models O_t^\star$, which is incorporation. Together the two establish Equation (1), and the proof is complete. □

Equation (13) is what event grounding, contract coverage and verifier validity are responsible for, and Appendix F measures how closely the derived pair tracks the oracle one under oracle-controlled events. What the corollary certifies is a property of every committed transition, which is the guarantee a runtime is asked for: nothing reaches the canonical graph unless it has been checked against the version it replaces. Liveness is supplied separately, by the budgets K_g and K_p of Algorithm 1.

The two inclusions carry different clauses. The hypothesis $\widetilde{W}_t \subseteq W_t^\star$ is sufficient and not necessary, and it is one side of a two-sided condition. It asks that the derived envelope grant no authority the oracle withholds, which is what lets preservation be inherited rather than checked against $W_t^\star$. The opposite inclusion carries the other clause: a certified patch writes only inside $\widetilde{W}_t$, so when S_t does not already satisfy $O_t^\star$, a certified commit can discharge the obligations only if the edits they require lie inside the envelope, $R_t^\star \subseteq \widetilde{W}_t$, which $W_t^\star \subseteq \widetilde{W}_t$ implies. The two inclusions hold together exactly when $\widetilde{W}_t = W_t^\star$. Appendix F measures the second inclusion under oracle-controlled events: the derived envelope contains the oracle region on every instance, $W_t^\star \subseteq \widetilde{W}_t$, with a mean excess of 1.6 entities, so every edit the obligations require lies within reach of a certified commit. Both clauses of Equation (1) are then scored directly against $(W_t^\star, O_t^\star)$.A grant wider than the oracle region does not by itself violate preservation, since membership permits a change without requiring one; what it does is move preservation from a certified property to a measured one.

C Runtime Implementation

Overview. The primary implementation of ASTER operates inside a ReAct-style actor loop (Yao et al., 2023). The actor reads the latest committed state, executes an environment-facing action, and receives feedback. Action outcomes, observations, user or peer updates, and environment changes can enter ASTER as candidate events. After an admitted event is processed, the actor continues from the latest committed state. On SWE-Amend, the same runtime interface is integrated into the Codex harness, which supplies the outer execution loop.

The runtime path is

$$e_t \;\to\; \textsc{Admit} \;\to\; \textsc{Ground} \;\to\; \textsc{Close}_\kappa \;\to\; \textsc{Propose} \;\to\; \textsc{Certify} \;\to\; \textsc{Commit}.$$

Within this revision path, model inference is used for grounding and patch proposal. Admission, propagation, shadow application, certification, and commit are runtime operations. The pseudo-code of ASTER are in Algorithm 1; prompts are provided in Appendix D.

C.1 Typed Hypergraph and Persistent State

Committed state. The runtime maintains the versioned state introduced in Equation (4),

$$S_t = (\nu_t, \mathcal{G}_t, L_t),$$

where ν_t identifies the committed version, $\mathcal{G}_t$ is the persistent typed hypergraph, and L_t records the accepted events and revisions that produced it. Entity identities persist across revisions, while execution-dependent attributes such as lifecycle state, resource holdings, and observation freshness can change.

Algorithm 1: ASTER transaction

Require: event e; budgets K_g, K_p; diagnostic $d \leftarrow \varnothing$

for $k = 1, \ldots, K_g$ **do**
  // I. Event grounding
  $S \leftarrow \mathrm{Snapshot}()$; $g \leftarrow \mathrm{Ground}(S, e, d)$;
  if g invalid **then continue**
  // II. Typed closure
  $(\widetilde{\boldsymbol{W}}_t, \boldsymbol{B}_t, \widetilde{\boldsymbol{O}}_t) \leftarrow \mathrm{Close}_\kappa(S, e, g)$
  for $j = 1, \ldots, K_p$ **do**
    // III. Bounded revision
    $\pi \leftarrow \mathrm{Propose}(\widetilde{\boldsymbol{W}}_t, \boldsymbol{B}_t, \widetilde{\boldsymbol{O}}_t, d)$
    // IV. Successor certification
    $(\bar{S}, d) \leftarrow \mathrm{Shadow}(S, \pi)$;
    if $\bar{S} = \perp$ **then continue**
    $(c, d) \leftarrow \mathrm{Certify}(S, \bar{S}, \widetilde{\boldsymbol{W}}_t, \widetilde{\boldsymbol{O}}_t)$
    if d stale **then break**; **if** $c = \perp$ **then continue**
    // V. Atomic commit
    if $\mathrm{Commit}(\nu(S), \bar{S}, c)$
      then return COMMITTED; **else break**
return DEFERRED, d

Task entities represent semantic behaviors, subgoals, or execution stages rather than individual model calls or environment primitives. A single task entity can therefore be realized through multiple actor actions. Results, resources, observations, and commitments occupy distinct semantic roles in the same persistent state.

Executable relations. Relations are directed typed hyperedges whose participants and attributes encode the execution structure maintained by the runtime. Hyperedges preserve joint conditions such as *all-of*, *any-of*, and k*-of-*n without reducing them to pairwise dependencies.

Each relation type is associated with the executable contract of Equation (5). The runtime implements the five contract families REQUIRES, DECOMPOSES, TEMPORAL, CONTENDS, and EVIDENTIAL. A domain schema binds its relation vocabulary and participant roles to these families together with the admissible entity roles, lifecycle transitions, attributes, and relation endpoint types. This binding is fixed during a transaction; only the instance graph evolves through committed revisions. The formal contract semantics are given in Table B.1.

C.2 GROUNDING AND RUNTIME PROPAGATION

Event admission. Before opening a transaction, the runtime checks whether the incoming feedback is already reflected in committed state or is otherwise known to require no state revision. Such events are bypassed and control returns to the execution loop. This gate provides idempotent handling of repeated events without requiring another graph revision. Other state-relevant feedback proceeds to grounding against a committed snapshot, which remains unchanged until a transaction successfully commits.

Event Grounding. External events are not mapped to graph entities through a benchmark-specific rule table. For a free-form event, the grounding call receives the event and the full committed hypergraph and returns a small set of persistent entities that the event directly affects:

$$g_t = \mathrm{Ground}(S_t, e_t).$$

The returned entities are seeds, not a writable region. The runtime validates their identifiers, types, and schema bindings before propagation. An invalid grounding does not modify committed state.

Runtime Propagation. Validated seeds initialize the runtime worklist. The runtime follows incident typed relations and evaluates their registered contracts against the unchanged snapshot S_t. Propagation continues until no additional affected entity or successor condition is produced:

$$(\widetilde{W}_t, B_t, \widetilde{O}_t) = \mathrm{Close}_\kappa(S_t, e_t, g_t).$$

Here $\widetilde{W}_t$ is the writable region, B_t is read-only boundary evidence, and $\widetilde{O}_t$ is the set of successor obligations.

Propagation is determined by the fixed relation semantics κ. A relation may propagate the event to another participant, emit a successor condition, or stop at a participant whose current state remains valid. Appendix B gives the corresponding closure properties and contract semantics. The completed closure defines the revision boundary before any patch is generated. The proposal stage can choose edits inside this boundary but cannot enlarge it.

C.3 Bounded Revision

Bounded Patch Proposal. The LLM model receives the event together with the completed revision context: $(e_t, \widetilde{W}_t, B_t, \widetilde{O}_t)$.

The model proposes one candidate patch π_t. The patch language contains node and relation additions, updates, and removals; relation updates can also revise their tail or head participants. Membership in $\widetilde{W}_t$ permits revision without requiring that every writable entity change. Boundary entities remain read-only, and observations or execution history provide context without expanding write authority. The model therefore chooses the concrete repair, while propagation, revision authority, and acceptance remain runtime-controlled.

Patch retry. A rejected candidate leaves committed state unchanged. Its diagnostic can be returned to the proposal call for another patch under the same snapshot, writable region, boundary, and obligations. Patch retries are bounded by K_p. If the transaction has to restart from a newer snapshot, grounding and closure are recomputed because the event interpretation and writable region are version-dependent.

C.4 Certification and Atomic Commit

Shadow Certification. Each candidate patch is applied to a fresh shadow copy of the base snapshot,

$$\bar{S}_t = \mathrm{Shadow}(S_t, \pi_t).$$

The runtime then checks the realized state difference rather than the patch's declared targets. Certification verifies that

$$\Delta(S_t, \bar{S}_t) \subseteq \widetilde{W}_t, \qquad \bar{S}_t \models \widetilde{O}_t,$$

together with the graph schema, lifecycle constraints, resource constraints, and registered domain invariants. A failed check rejects the candidate without changing the committed graph.

Atomic commit. A certified successor is committed only if the base version is still current. The commit atomically installs the successor, advances the version, and appends the accepted transition to L_t. This version check is the concurrency boundary. Centralized and distributed multi-agent executions may produce several events or revision requests against the same state. A candidate derived from a stale version cannot overwrite a newer commit. It is discarded and the transaction restarts from the current snapshot.

After a successful commit, the new state is the only committed hypergraph visible to subsequent transactions. Atomicity applies to the represented planning state. Environment actions, repository edits, and external tool effects are not rolled back; their resulting receipts can instead enter the host loop as later events.

D PROMPTS AND INSTRUCTIONS

This appendix records the model-facing instructions used in the experiments. The compared planning methods share the same ReAct execution protocol (Yao et al., 2023). FlatReplan uses an adapted plan–execute–replan prompt, while GraSP follows its released implementation and prompting procedure (Xia et al., 2026). ASTER introduces two additional model calls inside the runtime transaction: event grounding and bounded patch proposal.

D.1 SHARED REACT EXECUTION PROMPT

The primary comparisons use the same ReAct interaction loop (Yao et al., 2023). At each step, the actor receives the task, the current planning state, the accumulated interaction history, the latest environment feedback, and the available action interface. Only the planning state differs across methods.

Benchmark-native task instructions, action or tool descriptions, and response formats are preserved. The following gives the common prompt template; fields in angle brackets are instantiated by the corresponding environment.

Shared ReAct execution prompt

```
Task:
<task instruction>

Current planning state:
<planning state>

Interaction history:
<previous actions and observations>

Latest observation:
<environment feedback>

You have access to the following actions or tools:
<action interface>

Use the following interaction format:

Thought:
<reason about the current state and the next action>

Action:
<one valid environment action or tool>

Action Input:
<arguments, if required>

The environment returns an Observation after each action.
Continue the Thought--Action--Observation interaction until
the task is complete.

Final Answer:
<return only when completion is supported by the environment>
```

For ASTER, the planning-state field contains the latest committed hypergraph state. FlatReplan supplies its current ordered plan, and GraSP supplies its current dependency graph. These representations change the planning information available to the actor, not the environment-facing ReAct protocol.

D.2 Baseline Planning Prompts

FlatReplan. FlatReplan follows the plan–execute–replan pattern used in standard planning agents, with the replanning interface adapted from the LangGraph Plan-and-Execute implementation LangChain (2026b). The planner retains completed execution results and regenerates the ordered remaining plan when new feedback changes the valid continuation.

FlatReplan replanning prompt

```
Task:
<task instruction>

Current remaining plan:
<ordered plan>

Executed steps and results:
<completed steps and execution results>

Latest observation:
<environment feedback>

Update the plan according to the execution results and the
latest observation.

If the task is complete, return COMPLETE.

Otherwise, return an ordered plan containing only the steps
that still need to be executed. Preserve completed progress
and do not add completed steps back to the remaining plan.

Use the following format:

Remaining Plan:
1. <step>
2. <step>
...
```

The resulting remaining plan is supplied as the planning state in the shared ReAct prompt above.

GraSP. GraSP follows the released graph-execution and repair procedure of Xia et al. (Xia et al., 2026), including its dependency representation, execution verification, bounded local repair, and replanning fallback. We use the authors' released prompting and algorithmic structure, with only the benchmark-facing task, observation, and action fields instantiated for the common execution interface. We therefore do not introduce a separate rewritten GraSP prompt.

D.3 ASTER Runtime Prompts

Within an admitted ASTER transaction, the runtime uses model inference only for GROUND and PROPOSE (Appendix C). Propagation, shadow execution, certification, and atomic commit are runtime operations.

Event grounding. Grounding maps the external event to the persistent entities it directly affects. The model receives the event and the complete committed hypergraph. The returned entities initialize runtime propagation; they do not define the writable region.

Event grounding prompt

```
Event:
<runtime event>
```

```
Committed hypergraph:
<current committed hypergraph>

Identify the persistent entities directly affected by the
event.

Return only the direct seed entities. Do not propagate the
event through graph relations and do not determine the
revision region. Propagation and write authority are derived
by the runtime after grounding.

Use existing committed entity identifiers.

Return:
{
  "seed_entity_ids": [...],
  "rationale": "..."
}
```

The runtime validates the returned identifiers and then computes

$$(\widetilde{W}_t, B_t, \widetilde{O}_t) = \mathrm{Close}_\kappa(S_t, e_t, g_t).$$

Bounded patch proposal. After closure, the model receives the event and the derived revision context. It proposes a concrete graph update within the supplied writable region.

Bounded patch proposal

```
Event:
<runtime event>

Writable state:
<entities and relations in the writable region>

Read-only boundary:
<boundary witnesses>

Successor obligations:
<required successor conditions>

Allowed edit operations:
<edit schema>

Propose one patch that responds to the event and establishes
the successor obligations.

Modify only the supplied writable state. Treat the boundary
as read-only. Use only the declared graph-edit operations.

Return:
{
  "add_nodes": [...],
  "update_nodes": [...],
  "remove_node_ids": [...],
  "add_relations": [...],
  "update_relations": [...],
  "remove_relation_ids": [...],
  "rationale": "..."
}
```

Unused edit fields are returned empty. The candidate patch is applied to a shadow copy of the committed state and is not installed unless runtime certification succeeds.

Patch-retry feedback. A rejected candidate produces a diagnostic that is inserted into the next proposal request. The base snapshot and the closure derived from it remain fixed during an inner patch retry.

Patch-retry feedback

```
The previous patch was rejected.

Diagnostic:
<runtime diagnostic>

The base state, writable region, read-only boundary, and
successor obligations are unchanged.

Revise the patch to address the reported failure. Do not
expand the writable region.

Return a new patch using the same output format.
```

E Comparison Experiment Details

Here, we provide details of the LLM configurations, baseline implementations, and benchmark tasks used in our comparison experiments.

E.1 LLM configurations

The primary comparisons use `gpt-5.4-mini` with reasoning effort `low`. Within each benchmark, all compared planning methods use the same model and decoding configuration.

We additionally evaluate:

- **ALFWorld:** `gpt-5.4-mini low` and `xhigh` reasoning effort on the same 38 tasks.
- **ParaCook:** `GPT-5 high` and `Qwen3.8-27B`, with thinking disabled and enabled for `Qwen3.8-27B`.

These settings test robustness across reasoning levels and model backbones.

E.2 ASTER and Baseline Implementations

ReAct-based planning methods. The primary comparison evaluates four planning mechanisms under the same ReAct execution loop, their instructions are given in Appendix D.2:

- **ASTER.** Maintains planning state as a versioned typed hypergraph. Runtime events are processed through grounding, typed closure, bounded patch proposal, successor certification, and atomic commit. Implementation details are in Appendix C.
- **FlatReplan.** Maintains an ordered task list. When execution deviates from the current plan, it regenerates the remaining plan from the available execution state while retaining completed progress.
- **GraSP.** GraSP (Xia et al., 2026) maintains a structured dependency graph and performs local repair over the affected region. The model revises the supplied repair frontier, with full replanning used when the local repair budget is exhausted.
- **Asgard-Base.** On AsgardBench, we additionally evaluate Asgard-Base, the benchmark's released full-replanning scaffold. It runs in the native AsgardBench harness and serves as the benchmark-specific replanning reference.

Codex harness. We use the Codex harness in two ways:

Type	Examples	Stage 1	Stage 2	#
Functional extension	Composition or nesting behavior; type coverage; input-boundary and exception handling; feature–configuration compatibility; output representation and semantic preservation	Full original request	Additional functional requirement	34
Documentation extension	Documentation of the implemented behavior, compatibility guarantees, configuration semantics, or task-specific usage	Full original request	Additional documentation requirement	33
Delayed requirement	A requirement already belonging to the intended task is withheld initially	Partial original request	Remaining original requirement	33
Total				**100**

Table E.1: SWE-Amend task construction. The 100 sampled SWE-bench Verified instances are split approximately evenly across three amendment types. Each task is executed as one continuous two-stage episode.

- **Unmodified Codex.** Codex retains its native planning mechanism and is reported as a reference system.
- **Plugins in Codex.** On SWE-Amend, ASTER, GraSP, and FlatReplan are integrated at the Codex planning or runtime interface. The outer Codex execution loop, repository tools, and execution environment remain unchanged, while the planning-execution mechanism is replaced by the corresponding method.

E.3 SWE-Amend Construction and Evaluation

SWE-Amend is built from 100 instances uniformly sampled from the 500 SWE-bench Verified instances (Jimenez et al., 2024; OpenAI, 2024). We divide the 100 instances as evenly as possible among three amendment types: 34 functional extensions, 33 documentation extensions, and 33 delayed-requirement tasks.

Two-stage construction. Every task is presented in two stages. For functional and documentation extensions, Stage 1 exposes the complete original SWE-bench Verified request. After the agent has started modifying the repository, Stage 2 introduces an additional requirement. For delayed-requirement tasks, Stage 1 exposes only part of the original request, and Stage 2 reveals the withheld requirement. The repository workspace and conversation are kept across the two stages, so the second stage is a continuation of the same coding episode rather than a fresh restart.

Continuation protocol. The Stage 2 amendment is provided as a task update after the first editing round. No repository state is reset: files, previous edits, tool state, and conversation history remain available. All compared methods receive the same Stage 1 task, the same Stage 2 amendment, and the same underlying repository state.

Evaluation. The submitted patch is evaluated only after Stage 2. It has to satisfy both the original SWE-bench Verified requirements and the requirements introduced or revealed in Stage 2. Functional amendments are checked by corresponding tests, while documentation amendments are checked by their task-specific acceptance criteria. Evaluator-side checks are not exposed to the agent. An instance counts as solved only when the complete two-stage specification passes.

E.4 Task Cohorts

Table E.2 summarizes the task cohorts used in our experiments.

Table E.2: Task cohorts used in our experiments.

Benchmark	Tasks	Cohort
ALFWorld	134	Full unseen evaluation set; a 38-task subset from `pick_cool_then_place_in_recep` and `pick_two_obj_and_place` is used for ablation studies.
ScienceWorld	144	Uniformly sampled from the released task set.
AsgardBench	54	Full non-dirty (`non-d`) split.
ParaCook	72	Uniformly sampled after generating the full environment set with a fixed seed.
PARTNR	100	Uniformly sampled from the released task set.
τ^2-bench	266	Randomly sampled from the telecom domain.
WebShop	300	Uniformly sampled from the released task set.
AutomationBench	60	Sales-domain cohort.
SWE-Amend	100	Uniformly sampled from SWE-bench Verified.

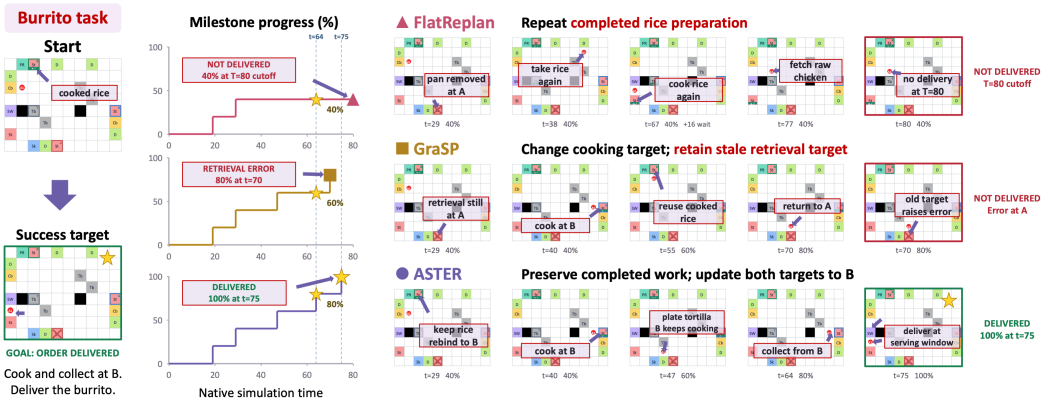


Figure 10: Recorded ParaCook execution trace. FlatReplan repeats completed rice preparation and reaches the $T = 80$ cutoff at 40% milestone progress. GraSP updates the cooking target to B but retains the stale retrieval target at A, reaching 80% before failing at the old target. ASTER preserves the completed rice, updates both affected targets to B, and delivers the order at $t = 75$. Curves show milestone completion over native ParaCook simulation time.

E.5 Recorded ParaCook Execution Trace

Figure 10 shows a representative recorded execution from the ParaCook evaluation cohort. The task requires completed work at station A to be reconciled with the remaining cooking and retrieval targets at station B. The trace makes the distinction between incorporation and preservation concrete: the runtime change needs to propagate to the remaining dependent state without discarding completed work that remains valid.

F Controlled Diagnostics of Transition Integrity

End-to-end task success does not identify why a state revision failed. A revision may leave a consequence of the new event unresolved, or it may discard committed state that the event did not affect. Definition 2.1 separates these errors as *incorporation* and *preservation*. We therefore construct oracle-controlled transitions for which both the required successor conditions and the legitimate revision boundary are known. The diagnostic first asks whether different boundary rules identify the right revision scope, then asks what transactional enforcement contributes once a boundary has been derived. Finally, we remove individual runtime mechanisms under compounding and concurrent events to find the regimes in which their removal costs integrity.

F.1 ORACLE-CONTROLLED DIAGNOSTIC

Instances and oracle annotations. Each instance contains a partly executed plan, a runtime event arriving mid-execution, and two oracle annotations: the ideal writable region $W_t^\star$ and the successor obligations $O_t^\star$. We write $R_t^\star \subseteq W_t^\star$ for the entities whose revision is required to satisfy those obligations. The oracle annotations are used only for evaluation and are never exposed to the agent.

The frozen diagnostic contains 150 instances from four plan families: cooking, search, delegation, and long-chain execution. Every configuration receives the same committed state S_t and event e_t, and uses `gpt-5.4-mini` with `reasoning_effort=low`. Each instance is repeated three times, giving 450 transitions per configuration.

Nested revision regions. Five nested regions organize the diagnostic:

$$W_t^{\mathrm{loc}} \subseteq W_t^\star \subseteq \widetilde{W}_t \subseteq W_t^{\mathrm{u}} \subseteq U_t. \tag{14}$$

Here W_t^{loc} contains only entities directly named by the event, $W_t^\star$ is the oracle boundary, $\widetilde{W}_t$ is the envelope produced by typed closure in Equation (6), W_t^{u} propagates through incident relations without typed stopping conditions, and U_t is the complete graph. Membership grants permission to revise an entity; it does not require that the entity change.

The derived envelope contains the oracle region on all 150 instances and matches it exactly on 37. Its excess over the oracle boundary is 1.6 entities on average. Thus every oracle-required edit lies inside the derived envelope, while the diagnostic can still expose unnecessary edits in $\widetilde{W}_t \setminus W_t^\star$.

Table F.1 gives the six admission conditions; every instance satisfies all six.

Table F.1: Diagnostic admission conditions. All 150 instances satisfy all six conditions. Gaps are measured in entities.

	Condition	Role in the diagnostic
C1	$W_t^{\mathrm{loc}} \subsetneq W_t^\star$, gap ≥ 2	separates direct event locality from oracle authority
C2	$W_t^\star \subsetneq W_t^{\mathrm{u}}$, gap ≥ 2	separates oracle authority from untyped propagation
C3	committed work in $W_t^{\mathrm{u}} \setminus W_t^\star$	makes over-revision observable beyond the oracle boundary
C4	committed work in $R_t^\star$	places previously committed state among required revisions
C5	$W_t^{\mathrm{u}} \subsetneq U_t$	distinguishes untyped reach from whole-graph revision
C6	$R_t^\star \subseteq W_t^\star$	keeps every required revision within oracle authority

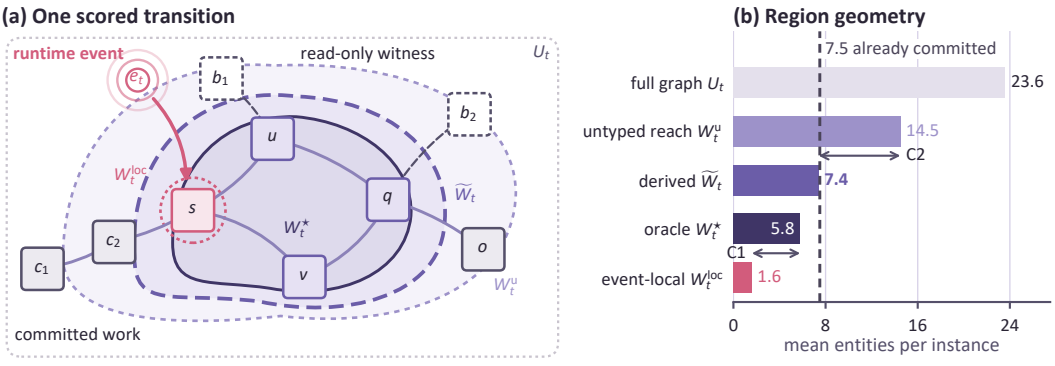


Figure 11: Oracle-controlled diagnostic. **(a)** An event arrives at a partially committed plan. The oracle boundary separates state legitimately open to revision from committed state that remains protected; the runtime independently derives $\widetilde{W}_t$ by typed closure. **(b)** Mean sizes of the five nested regions over the 150 instances. The dashed line marks committed work present when the event arrives; C1–C2 mark the admission gaps of Table F.1.

Table F.2: Single-event controlled diagnostics, 150 instances × 3 repeats. **(a)** Boundary study, Codex CLI held fixed; Gate marks commit-time enforcement of the stated region. **(b)** Transaction study; Commit gives the non-empty-commit rate and integrity among those commits. I/U/O/B counts integrity, under-, over- and both-violated transitions; n_{10}/n_{01} are discordant counts against full ASTER, descriptive only. Bold marks the highest TIR in (b).

(a) Revision boundary

Arm	Gate	Transition integrity			Scope		I/U/O/B	Empty
		Inc.	Pres.	TIR	P	R		
Derived envelope $\widetilde{W}_t$	no	.960	.978	.938	.993	1.000	422/18/10/0	0
Unrestricted	—	.864	.833	.702	.937	.868	316/59/73/2	0
Full graph U_t	yes	.882	.802	.698	.926	.894	314/47/83/6	0
Untyped reach W_t^{u}	yes	.822	.769	.613	.917	.837	276/70/94/10	0
Event-local W_t^{loc}	yes	.000	1.000	.000	.991	.000	0/450/0/0	4
Reference without maintained structured state								
Unstructured model	—	.929	.776	.716	.917	.936	322/27/96/5	0

(b) Transactional enforcement

Arm	Commit		Transition integrity			Scope		I/U/O/B	Empty	Discordant
	Rate	Int.	Inc.	Pres.	TIR	P	R			n_{10}/n_{01}
ASTER	.791	.961	.780	.980	**.760**	.785	.789	342/99/9/0	94	—
Codex host	.818	.878	.738	.978	.718	.811	.803	323/117/9/1	82	37/18
no scope narrowing	.736	.949	.718	.980	.698	.731	.732	314/127/9/0	119	52/24
single attempt	.340	1.000	.340	1.000	.340	.340	.340	153/297/0/0	297	191/2

Scoring transition integrity. For the realized successor S_{t+1}, let Δ_t denote the entities affected by its edits. We score the two clauses of Definition 2.1 directly against the oracle:

$$I_t = \mathbf{1}[S_{t+1} \models O_t^\star], \qquad P_t = \mathbf{1}[\Delta_t \subseteq W_t^\star], \qquad T_t = I_t P_t. \tag{15}$$

Their means are the incorporation rate, preservation rate, and transition-integrity rate (TIR), respectively. The pair (I_t, P_t) gives four mutually exclusive outcomes: *integrity* $(1,1)$, *under-revision* $(0,1)$, *over-revision* $(1,0)$, and *both violated* $(0,0)$.

A transition that installs no revision—including a deferred transaction, a no-edit proposal, or an unparsable proposal—is recorded as *Empty*. It remains in the denominator and scores $(0,1,0)$. We additionally report scope precision against $W_t^\star$ and recall against $R_t^\star$, with an empty edit scoring zero.

The single-event study contains two controlled comparisons. The *boundary study* fixes the Codex CLI proposer and changes the revision region made available to it. The *transaction study* starts from the full ASTER runtime and changes the proposer, scope narrowing, or feedback-guided retry. Table F.2 reports these comparisons in separate panels.

F.2 Revision Boundary

The boundary study asks where propagation should stop. The Codex CLI is held fixed while the available revision region changes from event-local state to increasingly broad regions. No successor-obligation certification or feedback-guided retry is used in this comparison. For the event-local, untyped-reach, and full-graph configurations, the stated region is enforced by the write-authority gate. The unrestricted configuration has no region restriction.

The derived-envelope configuration is intentionally different: $\widetilde{W}_t$ is supplied to the proposer as typed boundary information but is not enforced at commit. It therefore measures how useful the boundary is as revision guidance in the absence of transactional enforcement, rather than serving as a certified ASTER configuration.

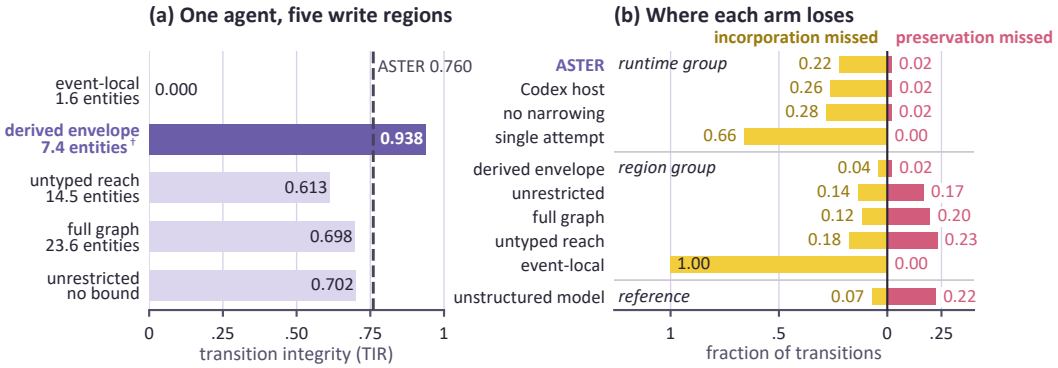


Figure 12: Revision regions and transition-integrity clauses. **(a)** The Codex CLI under five boundary configurations; † marks the derived envelope supplied as guidance without commit-time enforcement. The dashed ASTER value is shown only as a runtime reference; its Codex-host configuration is reported separately in Table F.2. **(b)** Fractions of transitions failing incorporation and preservation.

Under- and over-revision separate at the boundary. The event-local region shows one extreme of the diagnostic: it preserves unaffected state on every transition but never incorporates all event consequences. Broader untyped regions recover substantially more incorporation, but at the cost of many unnecessary edits. Untyped reach, for example, reaches .822 incorporation but only .769 preservation.

Supplying the typed envelope produces a different profile. Even without a commit-time gate, the derived-envelope configuration reaches .960 incorporation and .978 preservation, with scope precision .993 and recall 1.000. The distinction is therefore not simply whether the model is given more or less state: typed closure identifies where propagation can stop while retaining the consequences that still need to propagate.

Region size alone does not determine preservation. Untyped reach is strictly contained in the full graph but preserves less (.769 versus .802). What differs is the composition of the region: W_t^{u} is obtained by propagating from the event seeds without the typed stopping conditions that produce $\widehat{W}_t$.

Typed relations locate the boundary errors. The oracle annotations also identify the relation at which an incorrect revision propagated too far or stopped too early. In the unrestricted, full-graph, untyped-reach, and unstructured configurations, `any`-mode REQUIRES is the most frequently crossed relation (Figure 13b). Its stopping condition depends on whether another admissible participant still satisfies the requirement, which connectivity alone does not encode.

The same diagnostics expose missed consequences. Under a single transaction attempt, unmet REQUIRES–`all` and `must_precede` consequences occur 289 and 110 times; under the full runtime these counts fall to 93 and 73. Figure 13c additionally reports variation across the three repeats.

F.3 TRANSACTIONAL ENFORCEMENT

The boundary study asks *where* a revision belongs. We next hold the derived boundary fixed and ask what the transaction contributes to what is committed.

Commit coverage and commit quality. When no revision is installed, TIR records an under-revision. It is therefore useful to separate the probability of producing a non-empty commit from the integrity of the commits that are produced:

$$\mathrm{TIR} = \underbrace{\frac{450 - \mathrm{Empty}}{450}}_{\text{commit coverage}} \cdot \underbrace{\frac{\mathrm{Integrity}}{450 - \mathrm{Empty}}}_{\text{integrity given commit}}. \tag{16}$$

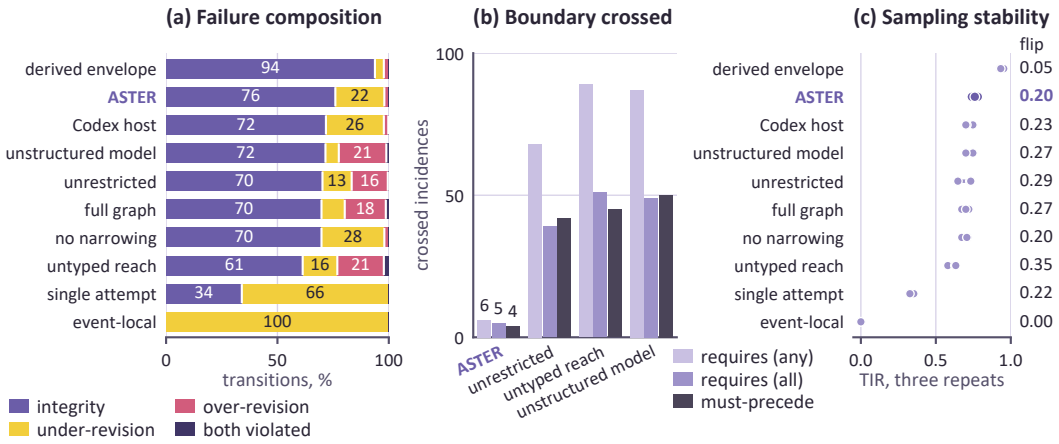


Figure 13: Failure structure and repeat variation. (a) The four transition outcomes, ordered by transition integrity. **(b)** Transitions crossing each typed relation boundary, counted once per relation kind. **(c)** TIR on each of the three repeats; the value at right is the fraction of the 150 instances whose outcome changes across repeats.

Full ASTER installs a non-empty revision on 356 of the 450 transitions, and 342 of those 356 commits satisfy both oracle clauses. Its commit coverage is therefore .791 and its integrity conditional on commit is .961, yielding TIR .760.

The single-attempt configuration exposes the role of feedback-guided retry. Its 153 commits all satisfy integrity, but 297 transitions install no revision. Restoring the full retry process adds 203 commits, 189 of which satisfy integrity, raising TIR from .340 to .760. Certification keeps uncertified candidates out of the committed state; retry turns rejections into certified successors.

Scope narrowing. Removing scope narrowing lowers TIR from .760 to .698 and lowers the all-transition scope precision and recall from .785/.789 to .731/.732. The change appears in both terms of Equation (16): the unnarrowed variant commits on fewer transitions and a smaller fraction of its commits satisfy both oracle clauses.

Authority enforcement. The write-authority gate constrains realized effects rather than relying on the patch's declared targets. Across configurations with an enforced region, no committed revision writes outside that grant. In the event-local configuration, the proposer attempts 188 out-of-region writes; none appears as an unauthorized committed edit. Preservation in Table F.2 is nevertheless scored against the stricter oracle region $W_t^\star$, so a revision committed legally inside $\widetilde{W}_t$ can still count as over-revision when it edits the excess $\widetilde{W}_t \setminus W_t^\star$.

Actor dependence. Replacing the native proposer with the Codex CLI while retaining the full transaction gives TIR .718 at preservation .978: the Codex host commits more often (.818 against .791) and its commits satisfy integrity less often (.878 against .961).

Transition integrity and goal reachability. Transition integrity is local to the revision from S_t to S_{t+1}; it does not require that the original task remain achievable after the event. The event itself makes the original goal unreachable on 237 of the 450 transitions. ASTER restores reachability on 28 of these and retains it wherever the event leaves it intact. Incorporating an event's consequences and retaining a plan that can still produce the original goal are therefore distinct outcomes; the end-to-end quantity is reported in Section 4.

F.4 Mechanism Stress Tests

The preceding experiments isolate boundary quality and transaction control under one event. We finally test the mechanisms that become important when transition errors can compound or multiple writers act on evolving state.

A switchable reference implementation of Algorithm 1 removes one mechanism at a time: hyperedge encoding, typed stopping, or the version guard. The study contains three splits with one episode per instance: 400 single-event episodes, 160 episodes with three compounding events, and 240 episodes with two concurrent writers and a write conflict. The concurrent split additionally evaluates removing boundary enforcement while retaining successor obligations.

For each mechanism, the regime in which its removal was expected to cost integrity was registered before the runs. Hyperedges represent joint conditions and therefore matter wherever such a condition occurs. Typed stopping is expected to become increasingly important when propagation errors survive into later events. The version guard concerns revisions to evolving state and is expected to matter only with more than one event.

For the three integrity columns of Table F.3, n_{10} counts episodes in which only the full runtime satisfies transition integrity and n_{01} the reverse. For conflict avoidance, the same notation counts whether only one configuration avoids a conflicting committed write. The table reports raw two-sided exact McNemar p-values; interpretation uses Holm correction across the twelve tests.

Table F.3: Mechanism removals under targeted event regimes. Cells report n_{10}/n_{01} for the full runtime against the named removal, with the raw exact McNemar p-value beneath. The first three outcome columns score transition integrity; the last scores conflicting-write avoidance on the 240 concurrent episodes.

Removed	Registered prediction	Single	Compounding	Concurrent	Conflict avoidance
Hyperedge → pairwise	loses in every family with a joint condition	$199/3$ $< 10^{-4}$	$78/5$ $< 10^{-4}$	$149/3$ $< 10^{-4}$	$5/0$ .063
Typed stopping	loses only once errors accumulate	$17/9$ .169	$57/7$ $< 10^{-4}$	$16/4$ .012	$13/0$ $< 10^{-3}$
Version guard	loses only with more than one event	$14/9$ .405	$51/10$ $< 10^{-4}$	$16/3$ .004	$16/0$ $< 10^{-4}$

Mechanisms separate in their stress regimes. Replacing joint hyperedges with pairwise edges produces the largest integrity loss in every regime: $199/3$ on single events, $78/5$ under compounding events, and $149/3$ under concurrent writers. Joint conditions occur throughout the diagnostic families, so their semantics matter even before events accumulate.

Typed stopping and the version guard show a different pattern. Their single-event differences are small ($17/9$ and $14/9$), whereas the gaps increase to $57/7$ and $51/10$ under compounding events and to $16/4$ and $16/3$ under concurrent writers. After Holm correction, all three hyperedge integrity comparisons remain significant, as do the typed-stopping and version-guard comparisons under compounding events, concurrent writers, and conflict avoidance; the largest adjusted p among these comparisons is .047. The hyperedge conflict-avoidance comparison and the two single-event comparisons for typed stopping and the version guard do not separate significantly.

Concurrent writes. Concurrent episodes expose the commit controls directly. The full runtime commits no lost update and no conflicting write across the 240 episodes. Removing the version guard permits conflicting writes in 16 episodes (6.7%), and removing typed stopping permits them in 13 (5.4%). Removing boundary enforcement while retaining successor obligations permits conflicting writes in 239 episodes (99.6%). The version guard controls whether a revision is still based on the version being installed, while the write boundary controls where that revision may take effect.

Taken together, the controlled diagnostics separate three sources of transition integrity: typed closure determines the revision boundary, certification and retry govern which candidate successor becomes committed, and the individual runtime mechanisms become consequential under the event structures they encode.

G NOTATION

Table G.1: Symbols used in the paper. A star marks a reference quantity of the oracle semantics, never observed by a method; a tilde marks the counterpart the runtime derives. The subscript t is dropped where no second time occurs.

Symbol	Meaning
USED IN THE MAIN PAPER	
$S_t = (\nu_t, \mathcal{G}_t, L_t)$	committed state when event e_t arrives: version, plan, event log
$\mathcal{G}_t = (V_t, E_t, \alpha_t)$	typed attributed hypergraph: entities, relations, execution attributes
$h = (T_h, H_h, \tau_h)$	hyperedge with tail, head and relation type
ρ_r	contract of relation type r: activate, propagate, obligate, stop
κ	contract vocabulary (domain binding), fixed for a run
$W_t^\star, O_t^\star$	ideal writable region and ideal successor obligations
$\Delta_t = \Delta(S_t, S_{t+1})$	entities the realized edits affect, incidence changes included
g_t	grounding: interpreted change and its seed entities
$\widetilde{\boldsymbol{W}}_t, \widetilde{\boldsymbol{O}}_t$	derived writable region and derived obligations
$\boldsymbol{B}_t$	boundary witnesses, read-only; $\boldsymbol{B}_t \cap \widetilde{\boldsymbol{W}}_t = \varnothing$
$\pi,\ \bar{S}_t$	proposed patch and its shadow state $\mathrm{Shadow}(S_t, \pi)$
K_g, K_p	grounding and patch budgets of one transaction
USED IN THE APPENDICES ONLY	
U_t	entity universe $V_t \sqcup E_t$
$S\|_X,\ u(S)$	restriction to $X \subseteq U_t$; untyped projection keeping only the incidence sets $\iota(h) = \{h\} \cup T_h \cup H_h$
$\mathfrak{A}_t,\ \mathrm{Atoms}(X)$	edit atoms, and those whose support $\mathrm{supp}(\delta)$ meets X
$D(S, S')$	atoms by which two states differ; $\Delta_t = \mathrm{supp}\, D(S_t, S_{t+1})$
C_t, F_t	evidence scope and frozen set $U_t \setminus \widetilde{\boldsymbol{W}}_t$; $\boldsymbol{B}_t = C_t \cap F_t$
ϕ_h	guarded transfer operator of a relation instance, $\phi_h(Z) \subseteq \iota(h)$
$Z^{(k)}$	entities reached after k propagation steps from the seed $Z^{(0)}$; $Z^{(\infty)}$ is the closure
$W_t^{\mathrm{loc}},\ W_t^{\mathrm{u}}$	event-local region, and untyped reach with no stopping condition
$R_t^\star$	required-edit support, $R_t^\star \subseteq W_t^\star$
I_t, P_t, T_t	incorporation, preservation and integrity indicators, $T_t = I_t P_t$