TAPESTRY: ENABLING CONTINUOUS COLLABORATION ACROSS EPISODES IN EMBODIED MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Embodied multi-agent collaboration is inherently continuous: task execution and intervening environmental changes can reshape the world in which subsequent tasks are performed. Effective collaboration requires agents to accumulate, share, and reuse distributed historical information across episodes, rather than repeatedly starting from scratch. Yet existing embodied multi-agent benchmarks largely isolate episodes, while memory-centric benchmarks primarily study long-horizon historical reasoning within individual embodied tasks, leaving interdependent cross-episode collaboration underexplored. To address this, we construct the **Tapestry Benchmark**, to the best of our knowledge the first controlled 3D benchmark for continuous embodied multi-agent collaboration across interdependent episodes. It organizes tasks into continuous streams in evolving environments and provides a unified testbed for studying how embodied agents accumulate, share, and reuse distributed experience under evolving environments and cross-episode continuity dependencies, where earlier task completion, environmental changes, and distributed multi-agent experience jointly determine the context of subsequent tasks. To this end, we propose **Tapestry Reasoning**, which views the evolving environment as a tapestry woven from distributed observations, actions, state transitions, and execution outcomes across agents and episodes. **Tapestry Weaving (TW)** interweaves fragmented historical evidence into coherent cross-episode histories, while **Tapestry Evidence Grounding (TEG)** retrieves and grounds relevant historical evidence to interpret current instructions, infer implicit task-relevant information such as object identities, states, locations, and configurations, and translate these into executable decisions. Evaluations on the benchmark demonstrate that Tapestry Reasoning consistently outperforms existing methods under matched backbone and execution settings across diverse LLM backbones. Extensive experiments further demonstrate its robustness under historical and execution interference and validate the increasing difficulty introduced by the proposed continuity dependencies.

1 INTRODUCTION

In real-world embodied collaboration, tasks rarely occur in isolation. Agents operate in evolving environments: objects may be moved or modified, spatial configurations may change, and information acquired by one agent can become relevant to another later (Biber et al., 2005; Hawes et al., 2017; Lauri et al., 2017); task completion may be inspected or assisted before the next request, while users or other activities can further change the environment between tasks (Baraglia et al., 2016; Igbinedion & Karaman, 2024). Consequently, subsequent tasks may take place in contexts shaped both by earlier task completion and by environmental changes occurring between tasks, creating temporal and inter-agent dependencies. These dependencies become particularly challenging when instructions are incomplete, ambiguous, or inconsistent with the current environment, requiring coordination and determining which prior information remains relevant and how to use it (Tellex et al., 2011; Misra et al., 2016). Effective long-term collaboration therefore requires a coherent shared history in which distributed experience can be accumulated, exchanged, and reused as tasks and environments evolve.

These requirements highlight two closely related capabilities underlying effective long-term collaboration: within-episode multi-agent coordination and cross-episode experience reuse. Existing embodied

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.

multi-agent research primarily focuses on within-episode coordination, enabling communication (Patel et al., 2021; Foerster et al., 2016), responsibility allocation, joint planning (Gerkey & Matarić, 2004; Gombolay et al., 2013), and execution toward shared objectives (Savva et al., 2019; Szot et al., 2021; Li et al., 2023). However, these settings treat episodes as independent units with simple navigation or QA tasks (Zhang et al., 2024; Mandi et al., 2024; Kannan et al., 2024), confining interaction history to a single episode and limiting its impact on subsequent ones.

In parallel, embodied memory research studies historical experience for personalized recall, history-dependent navigation, and spatial reasoning (Kwon et al., 2025; Yadav et al., 2025; Hu et al., 2026), while spatial memory representations support scene-level referential grounding (Xie et al., 2024; Gu et al., 2024; Huang et al., 2023). However, these settings largely consider a single acting agent and do not explicitly model how experience distributed across multiple agents and earlier embodied tasks jointly determines the grounding and execution of later interdependent tasks. This leaves a distinct gap between within-episode collaboration and distributed cross-episode experience reuse: how can agents continuously organize, share, and reuse historical experience across interdependent episodes?

This gap is also reflected in existing embodied benchmarks, which lack a systematic setting for evaluating continuous collaboration across interdependent episodes. Most embodied multi-agent benchmarks treat episodes independently, with instructions, initial conditions, and objectives defined separately, limiting study of how experience acquired by one agent in earlier episodes can be shared with others or influence subsequent episodes. Memory-centric benchmarks may transfer historical experience, but later tasks need not depend on entities, states, or configurations established earlier, nor require evidence distributed across teammates. Consequently, existing evaluations cannot systematically assess the challenge of continuous collaboration: not simply retaining longer histories, but organizing, sharing, and reusing distributed experience as earlier episodes shape later contexts.

To address this gap, we introduce the Tapestry Benchmark, a controlled 3D setting for continuous embodied multi-agent collaboration across interdependent episodes, together with Tapestry Reasoning, a framework that weaves distributed multi-agent experience into cross-episode tapestries for subsequent decision-making. We evaluate it across diverse LLM backbones under varying memory, temporal, and interference conditions, assessing cross-episode experience reuse and robustness as historical reasoning demands increase. Together, they support systematic study of continuous collaboration through distributed experience. Our contributions are threefold:

(1) Benchmark. We construct **Tapestry Benchmark**, a controlled 3D benchmark for continuous embodied multi-agent collaboration across interdependent episodes. It organizes episodes into evolving task streams through agent-policy execution, failure correction, and scene-valid inter-task environmental changes, while observations, actions, outcomes, and communication accumulate as distributed experience for evaluating cross-episode experience accumulation, sharing and reuse.

(2) Framework. We propose **Tapestry Reasoning**, a framework that organizes experience distributed across agents and episodes into cross-episode histories of evolving tasks and environments. **Tapestry Weaving** interweaves fragmented observations, actions, state transitions, and execution outcomes across agents and episodes, while **Tapestry Evidence Grounding** retrieves and grounds evidence to interpret current instructions, infer task-relevant information, and produce executable decisions.

(3) Evaluation. We conduct systematic evaluations across various LLM backbones, memory demands, temporal horizons, and historical and execution interference conditions. The results show consistent gains from Tapestry Reasoning across these settings, while revealing how performance is affected by increasing memory demands and interference in continuous cross-episode collaboration.

2 Related Work

2.1 LLM-powered Embodied Multi-Agent Collaboration

Embodied multi-agent collaboration spans classical multi-robot coordination and LLM-driven collaboration. Classical methods focus on task allocation and spatio-temporal scheduling under task, capability, and resource constraints (Gerkey & Matarić, 2004; Gombolay et al., 2013), while LLM-based systems add natural-language understanding, task decomposition, and feedback-driven planning for coordination (Ahn et al., 2022; Huang et al., 2022; Liang et al., 2023). Recent work further studies within-episode collaboration: RoCo integrates dialogue, trajectories, and collision for manipulation

Table 1: Comparison with existing benchmarks. Tapestry Benchmark operates in continuous multi-room 3D scenes, supports high- and low-level execution, covers relative-spatial, execution-order, constraint-free, and mixed tasks, evaluates short-(within-) and long-term(cross-episode) memory, and tests robustness to similar objects, semantic perturbations, and erroneous or infeasible instructions.

Benchmark	Type	Continuity	Env.	Agents	Comm.	Action	Constraint	Memory	Robustness	Scale
C-WAH (Zhang et al., 2024)	MAC	✗	3D-M	2	✓	HL	✗	✗	✗	10 test episodes
TDW-MAT (Zhang et al., 2024)	MAC	✗	3D-M	2	✓	LL+HL	✗	✗	✗	24 test episodes
PARTNR (Chang et al., 2025)	MAC	✗	3D-M	2	✗	LL+HL	S/T/F	✗	✗	100K episodes
SMART-LLM (Kannan et al., 2024)	MAC	✗	3D-S	1–4	✗	HL	✗	✗	✗	36 tasks
Habitat-MAS (Chen et al., 2025)	MAC	✗	3D-M	≥ 2	✓	LL+HL	✗	✗	✗	519 episodes
Memory Maze (Pasukonis et al., 2022)	MEM	✗	3D-Maze	1	✗	LL	✗	ST	✗	4 tasks
MEMENTO (Kwon et al., 2025)	MEM	✗	3D-M	1	✗	HL	✗	LT	Obj.	438 episodes
FindingDory (Yadav et al., 2025)	MEM	✗	3D-M	1	✗	LL+HL	T	ST	✗	1578 episodes
3DMem-Bench (Hu et al., 2026)	MEM	✗	3D-M	1	✗	HL	✗	ST	✗	1,860 tasks
LMEE-Bench (Wang et al., 2026)	MEM	✗	3D-M	1	✗	LL+HL	✗	ST	✗	166 test tasks
SEER-Bench (Cai et al., 2025)	MEM	✗	3D-M	1	✗	LL+HL	✗	ST	Err.	480 tasks
TAPESTRY BENCHMARK	**MAC+MEM**	✓	**3D-M**	≥ 2	✓	**LL+HL**	**S/T/F/M**	**ST+LT**	**Obj.+Sem.+Err.**	**1,600 episodes**

planning (Mandi et al., 2024); SMART-LLM performs task decomposition, coalition formation, and allocation (Kannan et al., 2024); CoELA tracks task progress, scene states, and teammate beliefs under partial observability and communication constraints (Zhang et al., 2024); and CaPo and CoTS improve planning via progress-adaptive meta-planning and collaborative tree search (Liu et al., 2025; Zu et al., 2025). However, these methods mainly coordinate within an episode, with limited prior-experience use. Recent studies also consider environmental and agent constraints. PARTNR models spatial, temporal, and heterogeneous capability constraints (Chang et al., 2025); Habitat-MAS matches capabilities via Robot Resume (Chen et al., 2025); COMBO predicts joint-action outcomes with compositional world models (Zhang et al., 2025); and FLEET schedules agents under precedence and capability constraints (Rivera et al., 2025). Together, they advance communication, task allocation, capability reasoning, collaborative planning, and world modeling, yet remain largely episode-centric: experience from one episode is not explicitly maintained as persistent team context for later episodes. How multiple agents can jointly form, consolidate, share, and reuse distributed experience across interdependent episodes for subsequent grounding and execution remains open.

2.2 EMBODIED COLLABORATION AND MEMORY BENCHMARKS

Existing embodied benchmarks can be broadly categorized into multi-agent collaboration (MAC) and embodied memory (MEM). MAC benchmarks evaluate communication, task coordination, heterogeneous capabilities, spatial and temporal constraints, and collaborative execution (Chang et al., 2025; Chen et al., 2025). However, they focus on episode-local collaboration, where histories are reset or isolated across episodes, without evaluating how earlier experience influences subsequent collaboration. In contrast, MEM benchmarks study historical reasoning for spatial understanding (Hu et al., 2026; Liao et al., 2026; Rasheed & Kabtoul, 2026), navigation (Pasukonis et al., 2022; Yadav et al., 2025; Park et al., 2025), sequential decision-making (Cai et al., 2025; Kwon et al., 2025; Ginting et al., 2025), and history-dependent tasks (Wang et al., 2026; Zhang et al., 2026). Although these benchmarks may require long-horizon histories, they primarily consider single-agent settings and do not model how experience jointly accumulated by multiple agents persists across episodes to support future collaboration. Table 1 summarizes that existing benchmarks largely overlook continuity. MAC benchmarks lack persistent cross-episode collaboration, whereas MEM benchmarks lack continuous multi-agent collaboration over shared historical experience. The Tapestry Benchmark bridges these directions by introducing controlled cross-episode continuity, enabling earlier observations, actions, and environment changes to persist and influence subsequent collaborative tasks.

3 TAPESTRY BENCHMARK

3.1 OVERVIEW

We construct the **Tapestry Benchmark**, a controlled 3D benchmark for continuous embodied multi-agent collaboration across interdependent episodes, where continuity is established through persistent environment evolution. Each task is first executed by an embodied agent policy, and successful

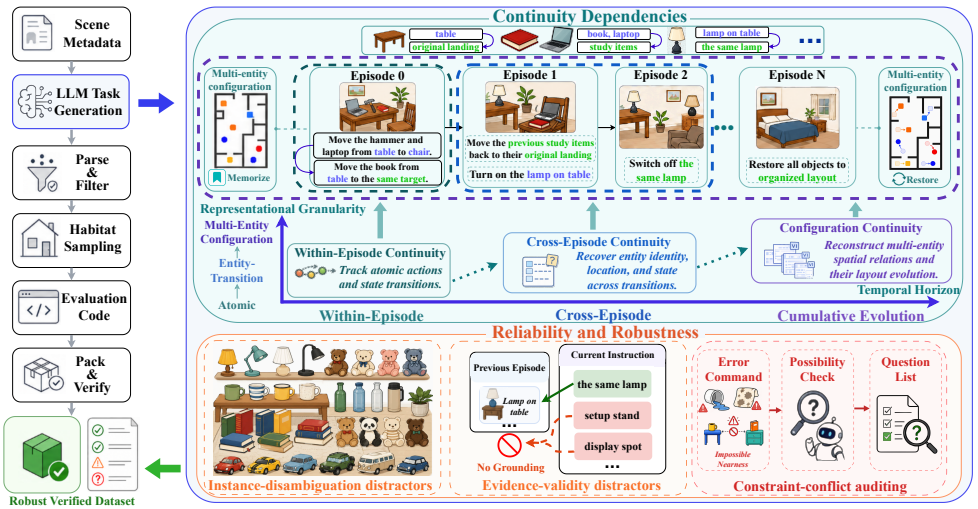

Figure 1: Overview of the Tapestry Benchmark. Starting from scene metadata, construction generates, compiles, and verifies interdependent multi-episode task streams in Habitat. The benchmark organizes continuous task streams with diverse historical dependencies across temporal horizons and representational granularities, while introducing controlled reliability challenges through instance-disambiguation distractors, evidence-validity distractors, and constraint-conflict auditing.

environment changes are retained. In practical embodied deployments, incomplete executions are typically resolved before subsequent tasks proceed, while environments may also evolve through external interventions, such as human activities, beyond the evaluated agents. Following this protocol, unresolved task requirements or invalid execution outcomes are manually completed or corrected, and additional scene-valid environment changes may be introduced before the verified environment state is propagated to the subsequent episode. Construction details are provided in Appendix D.

As shown in Fig. 1, construction proceeds through task-chain generation, compilation, and simulator-based verification. Given scene metadata, we generate multi-episode task streams. Rule- and simulator-based checks filter invalid candidates; the remaining chains are instantiated and verified in Habitat. The benchmark varies cross-episode dependencies along temporal span and representational granularity, ranging from within-episode process trajectories, through cross-episode references to persistent entities and states, to long-term evolution of multi-entity scene configurations. Five complexity levels extend historical horizons, representational granularity, and task constraints, demanding stronger historical and spatio-temporal reasoning and multi-agent collaboration. Together, these designs enable systematic evaluation of how embodied agents accumulate, share, and reuse distributed experience across interdependent episodes in evolving environments.

3.2 Continuity Dependencies

The benchmark organizes continuity dependencies across increasing temporal horizons, ranging from within-episode execution, through cross-episode interaction, to long-term environment evolution. These dependencies reflect how agents preserve and exploit historical information over time. **(1) Within-Episode Continuity.** Later subtasks may depend on observations, actions, execution outcomes, or object states established earlier in the same episode. Successful execution requires preserving local context throughout the trajectory so implicit requirements can be interpreted and satisfied conditions remain valid. **(2) Cross-Episode Continuity.** Cross-episode dependencies arise from the continuous evolution of entities, locations, and functional states across episodes. Subsequent tasks may omit explicit information of target entity, location, or state, requiring agents to recover targets from accumulated cross-episode experience, potentially from evidence distributed across agents. **(3) Configuration Continuity.** Configuration continuity extends cross-episode dependencies over longer horizons through evolving scene configurations. Each configuration records identities, locations, functional states, and spatial relations of entities. Intermediate episodes may alter parts of the configuration, while layout-restoration tasks require recovering a specified historical version.

3.3 Reliability and Robustness

Continuous embodied collaboration must remain reliable under ambiguity, noise, and execution uncertainty arising in real-world environments. Similar objects may coexist in the same scene, plausible descriptions may not correspond to actual history, and instructions may become incompatible with evolving environment states or physical constraints. To evaluate robustness against representative real-world interference at the levels of perception, historical reasoning, and execution, the benchmark introduces three controlled challenges. Instance-Disambiguation Distractors introduce visually similar or categorically identical objects, making category information alone insufficient for unique target identification. Agents must ground object references using location, appearance, state changes, and movement history. Evidence-Validity Distractors insert linguistically plausible but historically unsupported descriptions into otherwise valid tasks. Agents must reject unsupported historical evidence while preserving valid task references. Constraint-Conflict Auditing evaluates instructions that conflict with agent capabilities, spatial relations, current object states, or layout requirements. Agents must determine task executability and identify the source of the conflict.

3.4 Task Formulation and Evaluation

We formulate each task stream as a continuous sequence of interdependent cooperative Dec-POMDP episodes:

$$\mathfrak{T} = \left(\sigma, \{\mathcal{E}_e\}_{e=0}^{K-1}, \{s_{e,0}\}_{e=0}^{K-1}\right), \qquad \mathcal{E}_e = \left(\mathcal{S}, \{\mathcal{A}_i\}_{i=1}^{N}, \mathcal{T}, \mathcal{R}, \{\Omega_i\}_{i=1}^{N}, \mathcal{O}, \gamma\right). \tag{1}$$

Here, σ is the shared multi-room scene, $\mathcal{E}_e$ is the e-th episode, and $s_{e,0}$ its initial environment state. Successive initial states are constructed through agent-policy execution, verified failure correction, and scene-valid inter-task environmental changes, forming a cross-episode history in which earlier entities, locations, states, events, and relations may be referenced later. $\mathcal{S}$ is the global state space, N the number of agents, $\mathcal{A}_i$ and Ω_i the action and observation spaces of agent i, $\mathcal{T}$, $\mathcal{R}$, and $\mathcal{O}$ the transition, reward, and observation functions, and γ the discount factor. Each episode further has history-conditioned instruction I_e and structured evaluator $\mathcal{G}_e$, and contains an ordered sequence $\mathcal{U}_e = (u_{e,1}, \ldots, u_{e,K_e})$ of navigation and manipulation subtasks. Later subtasks may depend on observations, states, or execution outcomes established earlier in the same episode.

At step t of episode e, agent i selects actions from its local observation $o_{e,t}^i$, I_e, historical memory, and exchanged messages. Since I_e may implicitly refer to states, events, or relations established in preceding episodes and distributed across agents, the system grounds it as

$$g_e = \phi(I_e, M_{<e}, s_{e,0}), \tag{2}$$

where ϕ is the grounding function, g_e the grounded task specification, and $M_{<e}$ the historical memory accumulated before episode e. Agents then jointly act and communicate to complete $\mathcal{U}_e$ while satisfying the explicit and history-derived constraints in g_e, enabling evaluation of within-episode execution, cross-episode collaboration, and historical reasoning.

The benchmark is evaluated along five dimensions. Task Completion (TC) measures the fraction of task requirements that are successfully completed, while Success Rate (SR) measures the fraction of episodes in which the entire task objective is achieved. Task-Completion Steps measure embodied execution cost. Replanning Frequency measures long-horizon planning stability and the coordination cost of multi-agent replanning. Robustness evaluates reliability under instance-disambiguation distractors, evidence-validity distractors, and constraint-conflict auditing. We additionally record diagnostic statistics on memory usage, asynchronous coordination, communication, and system overhead to localize failures in historical grounding, collaboration, and execution.

4 Method

4.1 Pipeline

Unlike conventional episode-based embodied tasks, later episodes depend on historical observations, task states, execution outcomes, and environmental changes accumulated across preceding episodes and distributed across agents. Existing memory methods primarily reason over history

within individual tasks or flat records, whereas our setting requires integrating fragmented evidence distributed across agents and episodes under evolving environments. Consequently, recovering the historical context required for current tasks remains challenging. To address this, we propose Tapestry Reasoning, which organizes distributed historical evidence into coherent cross-episode histories. Tapestry Weaving (TW) interweaves fragmented multi-agent observations, actions, state transitions, and execution outcomes, while Tapestry Evidence Grounding (TEG) retrieves and grounds relevant historical evidence to infer task-relevant identities, states, locations, and configurations and translate them into executable decisions. As shown in Figure 2, Tapestry Reasoning operates alongside task decomposition, distributed planning, communication, and embodied execution, with new observations and execution outcomes continuously incorporated into shared memory for subsequent episodes.

4.2 Tapestry Weaving

In cross-episode task stream $\mathfrak{T}$, information from different agents is incomplete, asynchronous, and distributed over time. Each agent partially observes $\mathcal{E}_e$, while later episodes may depend on multiple teammates' operations and resulting state changes. Tapestry Weaving therefore interweaves distributed historical evidence into coherent cross-episode histories capturing the evolution of entities, states, and scene configurations across $\mathfrak{T}$. TW represents locations, attributes, and functional states from $M_{<e}$ as versioned entity nodes, while recorded interactions link them through actions, timestamps, and sources. Spatial relations, functional dependencies, and compositional structures further weave these records into evolving scene configurations across episode environments. The resulting histories provide a temporally grounded representation for retrieval and reasoning.

From this representation, historical retrieval must resolve four complementary ambiguities: *when* a state is valid, *how* it was established, under *what* surrounding configuration it remains consistent, and *which* entity it refers to. Accordingly, TW provides four coupled semantic views. The State-Validity View anchors recorded states to their temporal context. The Transition-Provenance View traces state transitions through recorded interactions and supporting evidence. The Configuration-Coherence View extends retrieved histories into temporally and spatially consistent multi-entity configurations. Finally, the Contrastive-Identity View distinguishes candidate entities with overlapping descriptions through interaction histories, state evolution, and movement trajectories. Together, these views preserve complementary evidence over entity identity, state version, and provenance.

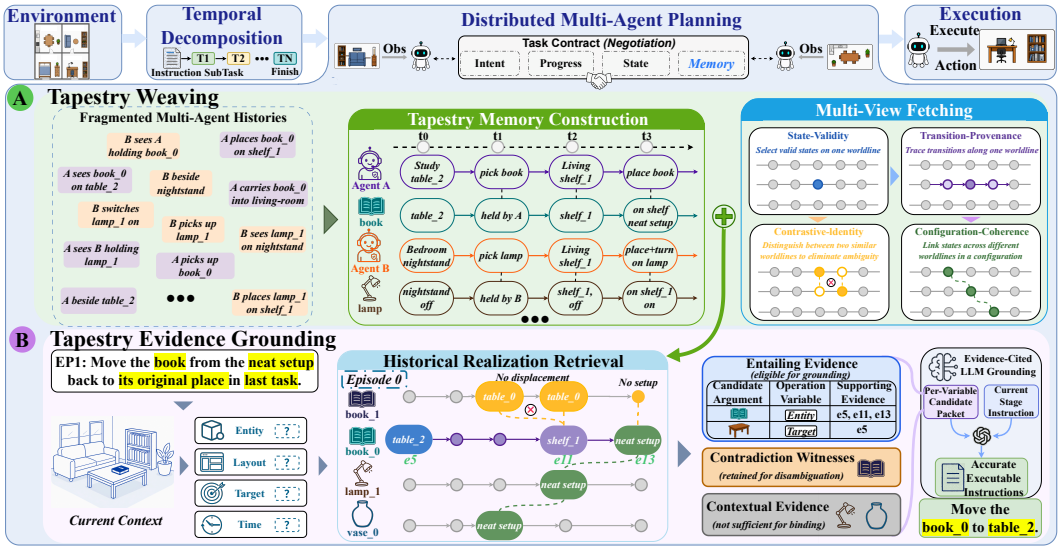

Figure 2: Pipeline of Tapestry Reasoning. **(A)** TW connects fragmented cross-agent and cross-episode histories into structured entity and configuration tapestries, supporting historical retrieval through four coupled semantic views. **(B)** TEG retrieves historical hypotheses from the tapestries and organizes them into an evidence package comprising three evidence types, distinguishing supported candidates from contradictory alternatives and auxiliary context for LLM grounding.

4.3 TAPESTRY EVIDENCE GROUNDING

Task instructions in real-world continuous task streams often do not specify all operational details required to complete tasks. Effective grounding therefore requires not only retrieving relevant tapestry evidence from $M_{<e}$ but also determining whether it supports the instruction and environment. TEG identifies unresolved operational variables from the instruction and environment and formulates them as typed retrieval conditions. It queries historical tapestry constructed by TW through semantic views, producing candidate hypotheses containing entity identities, state versions, evolution trajectories, relational contexts, and supporting evidence. Since retrieval conditions may not fully capture task semantics, multiple hypotheses may satisfy the same query. TEG therefore ranks candidates by evidence coverage before grounding. A candidate may represent a single entity or a jointly referenced entity group. Let $\mathcal{M}(O)$ denote the semantic commitments required by hypothesis O and $\mathcal{E}^+(O)$ denote commitments supported by retrieved evidence. The evidence coverage is:

$$c_t(O) = \frac{|\mathcal{E}^+(O) \cap \mathcal{M}(O)|}{|\mathcal{M}(O)|}. \tag{3}$$

TEG organizes retrieved evidence by grounding roles rather than treating all records equally. Candidate hypotheses conflicting with historical evidence or the current environment are grouped as Contradiction Witnesses. The top-ranked compact hypotheses with sufficient evidence coverage and their supporting records form the Entailing Evidence. Other candidates are retained as Contextual Evidence, providing auxiliary context without sufficient grounding support. This organization preserves alternatives while distinguishing supporting, conflicting, and contextual evidence. The LLM grounds instructions by jointly reasoning over these evidence types and the current environment. If no candidate is sufficiently supported, the corresponding dependency remains `UNRESOLVED`.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Follow previous works (Wang et al., 2026; Chang et al., 2025; Chen et al., 2025), we construct a balanced 300-episode evaluation suite from full 1,600-episode, comprising 60 from each of the five complexity levels. It covers diverse continuity dependencies, cross-agent information distributions, historical interference, and execution constraints. All methods use identical environment and execution settings across LLM backbones. Additional details are provided in Appendix C.

5.2 OVERALL PERFORMANCE

Table 2 compares Tapestry Reasoning with PARTNR (Chang et al., 2025) across LLM backbones. Under identical execution settings, Tapestry Reasoning improves task performance, showing the value of organizing and grounding cross-episode history. GPT-5.5 achieves the highest TC, SR,

Table 2: Overall performance across LLM backbones. PARTNR serves as baseline, relying on shared current-state graph for implicit joint execution without explicit modeling of cross-episode history.

Method	**Backbone**	**Task performance**				**Execution and planning**		**Impossible-task detection**	
		TC (%) ↑	**SR** (%) ↑	**ΔTC** (pp) ↑	**ΔSR** (pp) ↑	**Steps** ↓	**Avg. replans** ↓	**Exact Hit** (%) ↑	**Coarse Recall** (%) ↑
PARTNR	DeepSeek-V4	31.39	6.33	-27.22	-17.00	10429.40	41.50	0.00	0.00
	Qwen3-8B	24.82	5.33	-33.79	-18.00	**3874.25**	43.49	0.00	0.00
Ours	GPT-5.5	**58.61**	**23.33**	–	–	8894.56	**17.53**	**20.83**	63.04
	DeepSeek-V4	55.99	19.00	-2.62	-4.33	10876.40	18.45	14.58	**73.91**
	Qwen3-32B	45.51	12.67	-13.10	-10.66	9081.06	32.21	0.00	69.57
	Qwen3-8B	40.90	10.00	-17.71	-13.33	13808.72	26.21	4.17	45.65
	Llama-3.3-70B	36.12	9.67	-22.49	-13.66	10741.66	38.45	0.00	43.48
	Llama-4-Scout	33.47	7.00	-25.14	-16.33	8580.55	38.54	0.00	45.65
	Llama-3.1-8B	24.60	4.67	-34.01	-18.66	7625.38	41.92	0.00	36.96

Table 3: Ablation study for Tapestry Reasoning components on DeepSeek-V4. Hist.-Dep. denotes verified history-dependent subset. Res., Req-R, G-R, and Timeout denote binding resolution, correct historical request generation, historical target recovery, and task timeout rate, respectively.

	Overall (%)		Δ vs. w/o Both (pp)		Hist.-Dep. (%)		Historical Grounding (%)			Efficiency	
Configuration	**TC**	**SR**	**TC**	**SR**	**TC**	**SR**	**Res.**	**Req-R**	**G-R**	**Timeout (%)**	**Steps**
w/o TW & TEG	44.47	8.33	–	–	40.12	4.70	–	–	–	62.33	16.8k
w/o TEG	48.06	16.67	3.58	8.33	43.65	11.11	–	–	–	69.67	16.9k
w/o TW	49.49	16.33	5.01	8.00	42.13	11.11	65.87	18.76	9.95	31.67	12.2k
TW → Semantic Summary	48.74	14.67	4.27	6.33	43.16	8.55	32.08	26.58	7.28	39.33	13.6k
Full	**55.99**	**19.00**	**11.52**	**10.67**	**49.64**	**13.68**	**86.19**	**32.80**	**22.21**	**24.00**	**10.9k**

and Exact Hit, while DeepSeek-V4 obtains highest Coarse Recall, indicating stronger recognition of infeasible cases. Stronger backbones better ground evidence into executable decisions. For impossible-task detection, Coarse Recall measures infeasibility recognition, whereas Exact Hit requires correct conflict-type and object localization when applicable. Tapestry Reasoning improves conflict recognition, highlighting cross-episode grounding. However, the best SR is only 23.3%, showing long-horizon continuity dependencies and execution conflicts remain major challenges.

5.3 Ablation Study

As shown in Table 3, Removing TEG increases timeout rate and cost despite retaining retrieval ability, showing histories require executable grounding. Replacing TW with semantic summaries degrades performance, suggesting that semantic compression loses instance-, temporal-, and relational-level distinctions required for accurate retrieval. The complete framework achieves the best grounding accuracy, task performance, and execution efficiency, demonstrating that structured tapestry representation and evidence grounding are essential for robust cross-episode collaboration.

5.4 Robustness under Historical and Execution Interference

Instance Disambiguation and Evidence Validation. We evaluate instance-disambiguation and evidence-validation robustness from Sec3. As shown in Figure 3, end-to-end historical grounding requires retrieving, resolving, and validating evidence. The two settings exhibit distinct failure modes. Under same-type distractors, models often retrieve the correct category, but Exact Selection drops while Distractor Capture remains high, making instance resolution the main grounding error. In contrast, evidence-validation distractors introduce plausible but historically unsupported phrases. Stronger backbones reject them at False Need stage, whereas weaker models propagate errors

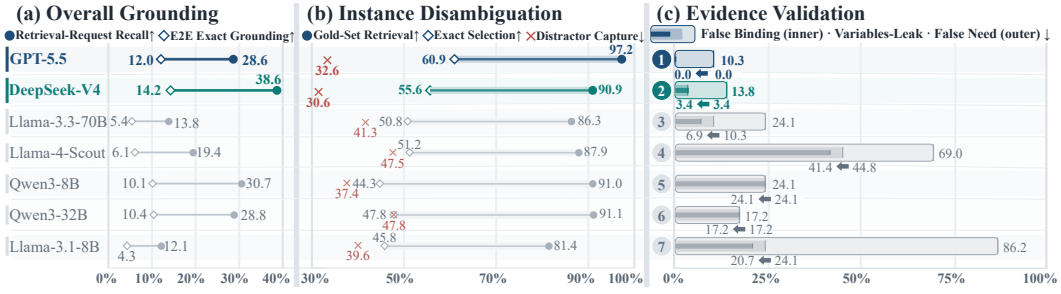

Figure 3: Robustness of historical grounding under *instance disambiguation* and *evidence-validation* interference. In (a), Retrieval-Request Recall counts correct requests for required historical dependencies regardless of request type (unlike Req-R) or retrieval results, while E2E Exact Grounding additionally requires exact target retrieval and selection. **(b)** Instance disambiguation under same-type distractors. Gold-Set Retrieval, Exact Selection, and Distractor Capture measure retrieval of the correct target set, exact final target selection, and erroneous selection of a same-type distractor, respectively. **(c)** Evidence validation under semantic-decoy distractors. False Need, Variables Leak, and False Binding measure unnecessary retrieval triggering, construction of unsupported historical variables, and unsupported entity grounding, respectively.

Table 4: Constraint-conflict auditing under different LLM backbones. TC, Zero-Prog., and Budget-Exh. characterize downstream execution after encountering infeasible constraints.

Metric	GPT-5.5	DeepSeek-V4	Qwen3-32B	Llama-3.3-70B	Qwen3-8B	Llama-4-Scout	Llama-3.1-8B
Infeasibility Diagnosis							
Exact Hit (%) ↑	**20.8**	14.6	0.0	0.0	4.2	0.0	0.0
Precision (%) ↑	90.6	64.2	52.5	80.0	48.8	84.0	**100.0**
Recall (%) ↑	63.0	**73.9**	69.6	43.5	45.7	45.7	37.0
F1 (%) ↑	**74.4**	68.7	59.8	56.3	47.2	59.2	54.0
FPR (%) ↓	0.16	0.99	1.51	0.26	1.14	0.21	**0.00**
Downstream Behavior							
TC (%) ↑	**54.1**	52.4	38.1	29.7	35.6	30.5	20.1
Zero-Prog. (%) ↓	**2.5**	5.0	**2.5**	10.0	7.5	10.0	30.0
Budget-Exh. (%) ↓	17.5	20.0	**2.5**	12.5	20.0	**2.5**	10.0

through Variable Leakage to False Binding. Compared with evidence validation, persistent failures in same-type instance resolution make instance disambiguation the more challenging robustness setting.

Constraint-Conflict Auditing. Table 4 evaluates constraint-conflict auditing and its impact on downstream execution. Models with broader conflict recognition achieve higher Recall, whereas precise localization is reflected by Exact Hit. These complementary capabilities correlate with better downstream task completion and fewer zero-progress failures. In contrast, lower budget exhaustion in weaker backbones often reflects premature task termination rather than more effective execution.

5.5 Performance Across Continuity Dependencies

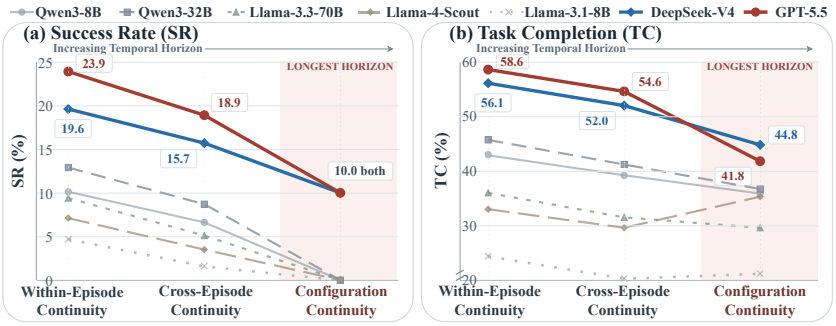

Figure 4: Performance across the three continuity dependencies defined in the benchmark: Within-Episode Continuity, Cross-Episode Continuity, and Configuration Continuity.

Figure 4 evaluates the three continuity dependencies in the benchmark. Across backbones, SR and TC decline from Within-Episode to Cross-Episode and Configuration Continuity, indicating increasing difficulty. Cross-Episode Continuity requires recovering task-relevant entities, locations, and states from accumulated history rather than recent context, while Configuration Continuity further reconstructs historically consistent multi-entity configurations across episodes, causing the largest degradation for most backbones. TC declines more slowly than SR, suggesting that agents often complete some required operations but fail to satisfy all history-dependent constraints for full success.

6 Conclusion

We presented Tapestry Benchmark for continuous embodied multi-agent collaboration across interdependent episodes in evolving environments together with Tapestry Reasoning, which interweaves distributed historical information into historical tapestry and grounds historical evidence for executable decisions. Experiments demonstrate consistent improvements in historical grounding, task performance, and execution robustness across LLM backbones. Our benchmark reveals that long-horizon continuity, fine-grained instance disambiguation, and constraint-conflict localization remain fundamental challenges for continuous embodied multi-agent systems.

AI USE STATEMENT

In this work, we used generative AI tools for the LLM Task Generation component of our dataset construction pipeline, where LLM APIs were employed to generate candidate simulated task data. These generated candidates were subsequently processed and validated through our dataset construction and verification procedures. We have not used generative AI tools for other tasks requiring disclosure. Additionally, we used generative AI tools for language polishing and improving the clarity and readability of the manuscript. We have reviewed all AI-assisted work. The authors take responsibility for the final content of this work, including the dataset, experimental results, analyses, claims, and manuscript text produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide detailed specifications to support reproduction of both the Tapestry Benchmark and Tapestry Reasoning. Sections 3–5 describe the benchmark formulation, continuity dependencies, method, evaluation metrics, and experimental protocol. Appendix B details the decentralized multi-agent execution framework, communication protocol, asynchronous planning and commit-time validation, and feedback-driven recovery procedure. Appendix C provides additional controlled experiments, ablations, efficiency analyses, prompt-reformulation studies, and evaluations on independently sampled suites. Appendix D documents the benchmark construction pipeline, including scene-grounded task generation, environment evolution, physical instantiation in Habitat, structured evaluator construction, human finalization, and episode- and stream-level verification. Appendix E provides the prompt interfaces and structured schemas used for action selection, historical-dependency routing, evidence grounding, and stage clarification. Together, these specifications make the benchmark construction, system execution, and reported evaluation procedures explicit and reproducible.

REFERENCES

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Jimmy Baraglia, Maya Cakmak, Yukie Nagai, Rajesh Rao, and Minoru Asada. Initiative in robot assistance during collaborative task execution. In *2016 11th ACM/IEEE international conference on human-robot interaction (HRI)*, pp. 67–74. IEEE, 2016.

Peter Biber, Tom Duckett, et al. Dynamic maps for long-term operation of mobile service robots. In *Robotics: science and systems*, pp. 17–24, 2005.

Zhongyi Cai, Yi Du, Chen Wang, and Yu Kong. Vision to geometry: 3d spatial memory for sequential embodied mllm reasoning and exploration. *arXiv preprint arXiv:2512.02458*, 2025.

Matthew Chang, Gunjan Chhablani, Alexander Clegg, Mikael Dallaire Cote, Ruta Desai, Michal Hlavac, Vladimir Karashchuk, Jacob Krantz, Roozbeh Mottaghi, Priyam Parashar, et al. Partnr: A benchmark for planning and reasoning in embodied multi-agent tasks. In *International Conference on Learning Representations*, volume 2025, pp. 65205–65268, 2025.

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*, volume 2025, pp. 56670–56690, 2025.

Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. *Advances in neural information processing systems*, 29, 2016.

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.

Muhammad Fadhil Ginting, Dong-Ki Kim, Xiangyun Meng, Andrzej Reinke, Bandi Jai Krishna, Navid Kayhani, Oriana Peltzer, David D Fan, Amirreza Shaban, Sung-Kyun Kim, et al. Enter the mind palace: Reasoning and planning for long-term active embodied question answering. *arXiv preprint arXiv:2507.12846*, 2025.

Matthew Gombolay, Ronald Wilcox, and Julie Shah. Fast scheduling of multi-robot teams with temporospatial constraints. 2013.

Qiao Gu, Ali Kuwajerwala, Sacha Morin, Krishna Murthy Jatavallabhula, Bipasha Sen, Aditya Agarwal, Corban Rivera, William Paul, Kirsty Ellis, Rama Chellappa, et al. Conceptgraphs: Open-vocabulary 3d scene graphs for perception and planning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5021–5028. IEEE, 2024.

Nick Hawes, Christopher Burbridge, Ferdian Jovan, Lars Kunze, Bruno Lacerda, Lenka Mudrova, Jay Young, Jeremy Wyatt, Denise Hebesberger, Tobias Kortner, et al. The strands project: Long-term autonomy in everyday environments. *IEEE Robotics & Automation Magazine*, 24(3):146–156, 2017.

Wenbo Hu, Yining Hong, Yanjun Wang, Leison Gao, Zibu Wei, Xingcheng Yao, Nanyun Peng, Yonatan Bitton, Idan Szpektor, and Kai-Wei Chang. 3dllm-mem: Long-term spatial-temporal memory for embodied 3d large language model. *Advances in Neural Information Processing Systems*, 38:67856–67884, 2026.

Chenguang Huang, Oier Mees, Andy Zeng, and Wolfram Burgard. Visual language maps for robot navigation. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10608–10615. IEEE, 2023.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*, pp. 9118–9147. PMLR, 2022.

Ifueko Igbinedion and Sertac Karaman. Learning when to ask for help: Efficient interactive navigation via implicit uncertainty estimation. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9593–9599. IEEE, 2024.

Shyam Sundar Kannan, Vishnunandan LN Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 12140–12147. IEEE, 2024.

Taeyoon Kwon, Dongwook Choi, Hyojun Kim, Sunghwan Kim, Seungjun Moon, Beong-woo Kwak, Kuan-Hao Huang, and Jinyoung Yeo. Embodied agents meet personalization: Investigating challenges and solutions through the lens of memory utilization. *arXiv preprint arXiv:2505.16348*, 2025.

Mikko Lauri, Eero Heinänen, and Simone Frintrop. Multi-robot active information gathering with periodic communication. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 851–856. IEEE, 2017.

Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, et al. Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *Conference on Robot Learning*, pp. 80–93. PMLR, 2023.

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 *2023 IEEE International conference on robotics and automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Chih-Ting Liao, Xi Xiao, Chunlei Meng, Zhangquan Chen, Yitong Qiao, Weilin Zhou, Tianyang Wang, Xu Zheng, and Xin Cao. Spamem: Benchmarking dynamic spatial reasoning via perception-memory integration in embodied environments. *arXiv preprint arXiv:2604.22409*, 2026.

Jie Liu, Pan Zhou, Yingjun Du, Ah-Hwee Tan, Cees G Snoek, Jan-Jakob Sonke, and Efstratios Gavves. Capo: Cooperative plan optimization for efficient embodied multi-agent cooperation. In *International Conference on Learning Representations*, volume 2025, pp. 70791–70815, 2025.

Zhao Mandi, Shreeya Jain, and Shuran Song. Roco: Dialectic multi-robot collaboration with large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 286–299. IEEE, 2024.

Dipendra K Misra, Jaeyong Sung, Kevin Lee, and Ashutosh Saxena. Tell me dave: Context-sensitive grounding of natural language to manipulation instructions. *The International Journal of Robotics Research*, 35(1-3):281–300, 2016.

Junyeong Park, Junmo Cho, and Sungjin Ahn. Mrsteve: Instruction-following agents in minecraft with what-where-when memory. In *International Conference on Learning Representations*, volume 2025, pp. 16922–16960, 2025.

Jurgis Pasukonis, Timothy Lillicrap, and Danijar Hafner. Evaluating long-term memory in 3d mazes. *arXiv preprint arXiv:2210.13383*, 2022.

Shivansh Patel, Saim Wani, Unnat Jain, Alexander G Schwing, Svetlana Lazebnik, Manolis Savva, and Angel X Chang. Interpretation of emergent communication in heterogeneous collaborative embodied agents. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 15953–15963, 2021.

A Haroon Rasheed and Maria Kabtoul. emem: A hybrid spatio-temporal memory system for embodied agents. *arXiv preprint arXiv:2606.03374*, 2026.

Corban Rivera, Grayson Byrd, Meghan Booker, Bethany Kemp, Allison Gaines, Emma Holmes, James Uplinger, Celso M de Melo, and David Handelman. Fleet: Formal language-grounded scheduling for heterogeneous robot teams. *arXiv preprint arXiv:2510.07417*, 2025.

Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, et al. Habitat: A platform for embodied ai research. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 9339–9347, 2019.

Andrew Szot, Alexander Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah Maestre, Mustafa Mukadam, Devendra Singh Chaplot, Oleksandr Maksymets, et al. Habitat 2.0: Training home assistants to rearrange their habitat. *Advances in neural information processing systems*, 34:251–266, 2021.

Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew Walter, Ashis Banerjee, Seth Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 25, pp. 1507–1514, 2011.

Sen Wang, Bangwei Liu, Zhenkun Gao, Lizhuang Ma, Xuhong Wang, Yuan Xie, and Xin Tan. Explore with long-term memory: A benchmark and multimodal llm-based reinforcement learning framework for embodied exploration. *arXiv preprint arXiv:2601.10744*, 2026.

Quanting Xie, So Yeon Min, Pengliang Ji, Yue Yang, Tianyi Zhang, Kedi Xu, Aarav Bajaj, Ruslan Salakhutdinov, Matthew Johnson-Roberson, and Yonatan Bisk. Embodied-rag: General non-parametric embodied memory for retrieval and generation. *arXiv preprint arXiv:2409.18313*, 2024.

Karmesh Yadav, Yusuf Ali, Gunshi Gupta, Yarin Gal, and Zsolt Kira. Findingdory: A benchmark to evaluate memory in embodied agents. *arXiv preprint arXiv:2506.15635*, 2025.

Haochen Zhang, Nirav Savaliya, Faizan Siddiqui, and Enna Sachdeva. Fast-eqa: Efficient embodied question answering with global and local region relevancy. In *2026 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pp. 1664–1673. IEEE, 2026.

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*, volume 2024, pp. 19373–19401, 2024.

Hongxin Zhang, Zeyuan Wang, Qiushi Lyu, Zheyuan Zhang, Sunli Chen, Tianmin Shu, Behzad Dariush, Kwonjoon Lee, Yilun Du, and Chuang Gan. Combo: Compositional world models for embodied multi-agent cooperation. In *International Conference on Learning Representations*, volume 2025, pp. 49996–50019, 2025.

Lizheng Zu, Lin Lin, Song Fu, Na Zhao, and Pan Zhou. Collaborative tree search for enhancing embodied multi-agent collaboration. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 29513–29522, 2025.

A Appendix Overview

The appendix contains three substantive parts, followed by the prompt templates used in our framework. Section B provides implementation details of the multi-agent execution framework, including system formulation, temporal decomposition, distributed coordination and communication, and feedback-driven recovery. Section C complements the main experiments with additional comparisons, collaboration analyses, mechanistic diagnostics, efficiency studies, robustness checks, and failure analysis; Table 5 summarizes these experiments and their corresponding sections. Section D details benchmark construction and validation, including task-stream formalization, generation, agent capabilities, continuity and reliability construction, evaluator verification, and dataset difficulty stratification. Finally, Section E provides the prompt templates used throughout the framework.

Table 5: **Overview of supplementary experiments.** Each row summarizes the question studied in the corresponding part of Section C and points to the primary supporting results.

Section	Supplementary experiment / analysis	Primary results
§C.1	**Alternative memory mechanisms.** Compares TW–TEG with alternative embodied-memory mechanisms under the same multi-agent execution pipeline to isolate the effect of structured cross-episode representation and evidence grounding.	Table 9
§C.2.1	**Single-agent vs. two-agent execution.** Evaluates whether distributed observations and collaboration improve task completion, and compares efficiency under increasing simulation-step and planning-call budgets.	Table 10; Fig. 7
§C.2.2	**Shared cross-agent historical information.** Removes cross-agent history sharing to measure how access to other agents' observations and experience affects historical grounding and downstream execution.	Table 11
§C.2.3	**Collaboration scaling.** Studies how task performance, communication cost, and individual agent contribution change as the team size increases.	Fig. 8; Table 12; Fig. 9
§C.3.1	**Historical retrieval and binding diagnostics.** Decomposes historical grounding into dependency triggering, retrieval, candidate discrimination, and final binding to identify where errors arise.	Tables 13 and 14
§C.3.2	**Execution throughput and planning overhead.** Compares execution efficiency across LLM backbones, including planning overhead and progress achieved under the same execution setting.	Table 15
§C.4.1	**Prompt-reformulation robustness.** Tests whether performance remains stable under semantics-preserving reformulations of task instructions rather than depending on a particular prompt wording.	Table 16
§C.4.2	**Evaluation-suite sampling stability.** Repeats evaluation on independently sampled suites to examine whether the reported performance trends are robust to evaluation-set composition.	Table 17
§C.5	**Representative failure analysis.** Traces a representative historical-reference grounding failure to illustrate remaining limitations in long-horizon retrieval, evidence discrimination, and binding.	Fig. 10

B MULTI-AGENT METHOD DETAILS

This section provides additional details on the cooperative execution layer surrounding the interface between Tapestry Weaving (TW) and Tapestry Evidence Grounding (TEG) in Tapestry Reasoning. We adopt decentralized distributed execution following the collaboration pattern of human teams in natural environments. Each participant acts from a local viewpoint, applies its own capabilities, and communicates task-relevant changes to the team. Accordingly, every agent owns a local planner and participates in a compact incremental coordination protocol. This organization aligns decision authority with embodied perception and action responsibility.

The resulting architecture uses an event-driven decentralized loop. Each agent plans from its current local context, exchanges incremental coordination updates, and rechecks a returned action against the latest team state before execution. The same loop incorporates executor feedback, advances the active task stage, and schedules replanning when new evidence changes an assignment or an action's feasibility.

B.1 SYSTEM BOUNDARY AND NOTATION

Let $\mathcal{A} = \{a_i\}_{i=1}^{N}$ be the configured team. Each agent has the same grounded skill interface $\mathcal{U}_i$, together with its own embodiment, local observation, role description, and action availability. The acting agent owns the planner that selects its high-level action from locally visible information. A runtime process coordinates simulation and message delivery across these agent-local planners.

Table 6 lists the implementation-visible state used throughout the execution loop.

Table 6: Notation is restricted to state that participates in a runtime decision.

Symbol	Meaning	Why it is required
G_i^t	Agent i's dynamic local world graph at decision event t	Supplies runtime-assigned instance identifiers for observed entities, together with observed states, rooms, and spatial relations under partial observability.
M_i	Agent i's local replica of provenance-tagged team episodic memory	Makes finalized experiences from every agent retrievable while preserving the source agent of each record.
$\mathcal{H}_i^t$	Evidence-grounded context produced through TW and TEG	Selects from shared historical records only when the active instruction contains an unresolved dependency.
Q_i^t	Validated teammate packets retained by sender at agent i	Provides the latest policy-visible intent, progress, held-object, and entity-delta evidence used for coordination.
f_i^t	Latest local executor response	Distinguishes success, temporary precondition failure, relation incompatibility, and other recovery conditions.
$\mathcal{C}_z^t$	Versioned execution specification for active stage z	Preserves requested operations, their partial order, explicit arguments and spatial/state constraints.
L_z^t	Completion record for stage z	Helps agents avoid duplicate work and supplies evidence for stage completion while allowing later observations to invalidate stale entries.

The superscript t denotes a logical decision event in an agent's runtime context. Local observation arrival, executor completion, packet reception, and planner return each create such an event. Events may overlap in wall-clock time across agents; the sender identifier, timestamp, and contract version provide the ordering information required for local assimilation and commit validation. At each decision event, agent i builds its planner context from the active stage, the current local world graph G_i^t, historical evidence $\mathcal{H}_i^t$ retrieved through TW–TEG when required, the latest validated teammate updates in Q_i^t, and the most recent executor feedback f_i^t. Only the latest relevant packet from each

sender is retained, keeping the context compact while preserving useful coordination updates. The local planner then proposes one grounded skill. Because planning is asynchronous, new observations, executor outcomes, or teammate updates may arrive during inference. Every returned proposal is therefore validated against the latest team state before execution, including the current coordination state, teammate intents, and completed effects. A still-valid proposal is executed, whereas a stale one is deferred and replanned from the updated state.

These inputs remain separated because they provide different forms of evidence. G_i^t represents the agent's current grounding of entities, states, rooms, and spatial relations; missing grounding triggers navigation or exploration before manipulation. $\mathcal{H}_i^t$ provides historical evidence only for unresolved dependencies, with explicit current observations taking priority and ambiguous bindings remaining unresolved. Q_i^t captures current team progress, such as teammate intents, held objects, relevant observations, and completed skills, while retaining sender and timestamp information so that stale updates can be replaced. Finally, f_i^t records the physical outcome of the latest attempted skill and supports recovery. This separation preserves the distinction between historical evidence, communicated team progress, current grounding, and direct execution feedback.

B.2 Temporal Decomposition and Stage Construction

Temporal decomposition is applied to instructions with either explicit temporal ordering constraints or multiple independent sentences. When an instruction contains explicit temporal expressions, we identify these constraints directly at the lexical level rather than relying on an LLM to infer latent ordering relations. The detected temporal relations, together with sentence and clause boundaries, are used to divide the instruction into ordered action segments. For instructions without explicit temporal constraints, we follow the natural language structure and split the instruction according to sentence boundaries, where each independent sentence is treated as a separate execution stage. This lightweight decomposition provides a simple and deterministic way to construct execution stages while reducing LLM invocations and inference latency.

Once the instruction is decomposed, each segment retains its original task content and forms an individual execution stage. For temporally constrained instructions, the relative order of segments is determined by the temporal relations expressed in the instruction. The next stage is activated when the LLM judges the current stage complete or when the runtime skips it after repeated unsuccessful attempts or prolonged execution. Skipping does not count as successful completion; task metrics remain determined by the evaluator.

B.3 Distributed Coordination and Incremental Communication

The distributed coordination protocol spans three timescales. At stage activation, the team initializes the active stage context. At episode finalization, agents replicate newly completed, provenance-tagged memory records. During embodied execution, intent, progress, and entity-delta packets synchronize the state required for concurrent action. These channels form a single coordination pipeline in which the assignment state defines current commitments, episodic synchronization distributes historical evidence, and online communication maintains execution consistency.

All three channels implement change-triggered incremental exchange. Assignment messages encode ownership or dependency revisions, memory synchronization appends newly finalized records, and online packets encode new intent, progress, or selected entity changes. Each update is transmitted once per sender–receiver pair and assimilated into sender-indexed state. This protocol controls communication redundancy and reserves planner-context capacity for information that can alter the next decision.

B.3.1 Provenance-Aware Shared Episodic Memory

During execution, agents retain the task instruction alongside provisional local traces of observations, actions, outcomes, and communication; finalized episodic records are synchronized after episode completion. Memory synchronization then appends this newly finalized record once to the queue of every other agent. A synchronized record is marked as received through memory sharing and stores both the source agent and the local memory owner. The episode–source key provides idempotent delivery.

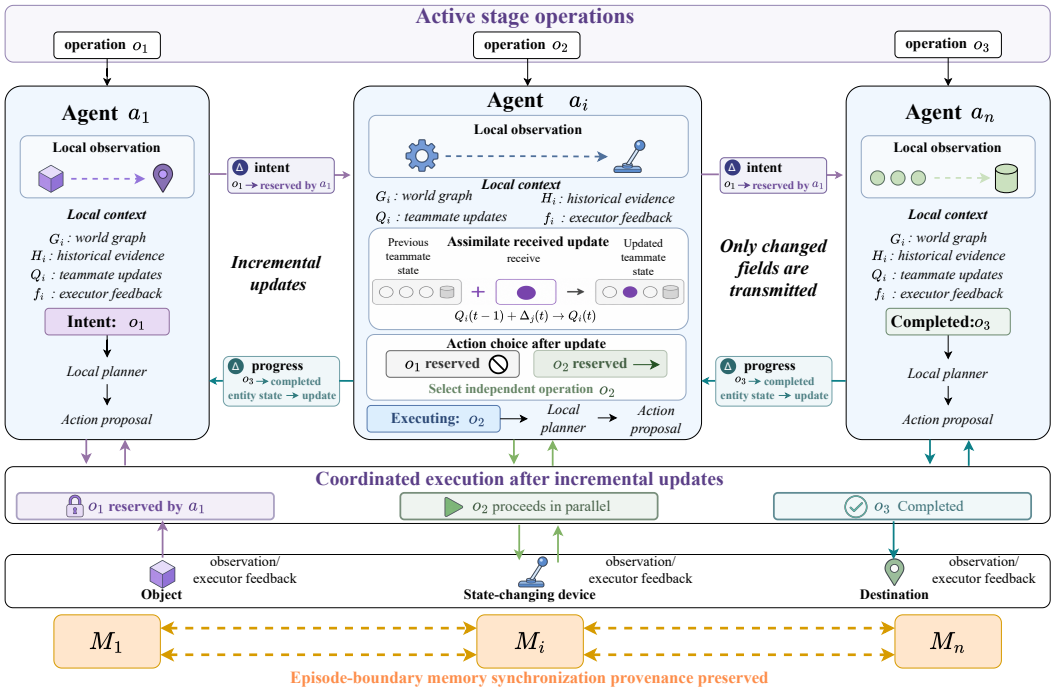

Figure 5: Overview of decentralized incremental coordination. Each agent maintains a local execution context and exchanges only changed intent, progress, and other updates. Incremental synchronization enables coordinated parallel execution while preserving local world representations.

The result is a provenance-aware replicated episodic memory. Tapestry Weaving (TW) connects the synchronized records into queryable entity and configuration tapestries; TEG identifies unresolved dependencies, retrieves relevant evidence, and binds only evidence-supported values. Source metadata remains attached throughout this process, so shared memory increases coverage while preserving who observed or executed the event.

Each synchronized memory record is scoped by its episode identifier, source agent, and recorded state transition. TW incorporates that record into the corresponding tapestries, while TEG may use the resulting evidence to resolve a historical dependency. Current manipulation arguments receive grounding from G_i^t and recent online updates. The separation between replicated history and agent-local current state preserves temporal validity during embodied execution.

B.3.2 Execution-Time Communication

Agents coordinate through typed incremental packets. A semantic update is serialized under a versioned protocol header containing a message identifier, sender, separately addressed recipient, message type, timestamp, and checksum. Header–payload consistency and integrity are checked before the message reaches the receiving planner. Table 7 summarizes the information exchanged in these incremental packets and the role of each field in coordination. Figure 5 illustrates this incremental coordination process, where each agent preserves its local world representation and exchanges only changed coordination fields.

On a useful intent or progress change, the sender constructs one incremental semantic update and transmits a separately addressed packet to every other configured agent. Each receiver validates the packet, applies the changed fields to its sender-indexed cache, records the received event in its local event history and applies grounded entity deltas to its coordination view. A planning call continues with the snapshot captured at its submission time; later packets participate in commit validation and the subsequent replan.

World graphs represent observed effects after skill execution. Concurrency control additionally requires prospective state. Committing a state-changing action therefore emits an intent packet that

Table 7: Incremental coordination schema. The planner sees a compact textual rendering of newly changed fields, while protocol handling uses the typed records.

Packet field	Content	Coordination function
Type and identity	Intent or progress; sender, recipient, message identifier, schema version	Supports validation, traceability, sender-indexed retention, and explicit fanout.
Completed skill	Skill name, grounded arguments, success status, executor response, and optional error	Updates the completion record or exposes a concrete repair condition.
Next intent	Optional next skill and grounded arguments	Reserves prospective state-changing work before its effect appears in a world graph.
Held object	Exact object currently carried by the sender	Warns other agents and discourages competing actions.
Entity deltas	A bounded set of target-object and target-location states and relations	Shares task-critical changes while the complete world graph remains local.
Stage context	Readiness, ownership, or progress fields changed since the previous report	Aligns local stage views and invalidates obsolete assignments through changed contract fields.
Memory synchronization	Newly finalized episode record, source agent, receiving memory owner, and synchronization marker	Incrementally extends team experience while preserving the existing queue and the historical status of each record.
Bandwidth metadata	Incremental flag, applied word budget, and change-selection status	Makes omission explicit, suppresses unchanged content, and keeps communication cost independent of raw memory size.

establishes a short-lived reservation over the manipulated objects, optional reference objects and held objects. The corresponding progress packet later records the executor outcome, relevant entity deltas, and reservation release. Together, the intent and progress fields cover the state immediately before and after an embodied effect.

Packet delivery follows the receiver's planning state. An idle receiver renders validated changes into its next prompt immediately. A receiver with an active planner call stores them in its pending queue and applies them during validation of the returned proposal.

B.3.3 Execution Coordination and Conflict Avoidance

During embodied execution, agents may act concurrently under partial observability, where independent actions can proceed in parallel while conflicting operations will be explicitly specified in the prompts of subsequent planning stages. To maintain physical consistency, the runtime performs coordination checks before executing each state-changing action. These checks consider the action's grounded arguments, currently observed environment state, recent teammate updates, and active temporal constraints.

Before execution, each proposed action is evaluated against the resources it may affect, including manipulated objects, destination locations, spatial anchors, and directly related workspace regions. Two actions are considered conflicting when they modify overlapping resources with incompatible effects. For example, simultaneous manipulation of the same object, movement of an object toward incompatible destinations, or operations violating an established spatial relation are explicitly specified in the prompts of subsequent planning stages. In contrast, actions affecting independent entities or compatible regions can proceed concurrently.

Execution-time coordination relies on incremental intent and progress packets. When an agent initiates a potentially state-changing operation, it broadcasts an intent containing the grounded object, target location, and expected effect. Teammates incorporate this information into their local

coordination context and avoid generating incompatible actions. After execution, the agent reports the resulting progress and entity-level changes, allowing other agents to update their local views and discard obsolete assumptions.

The runtime filters each agent's available skills according to current observations, teammate updates, historical grounding results, and active execution constraints. Manipulation actions are considered only when their required entities are grounded and their effects are compatible with the latest coordination state. When necessary, navigation, observation, or waiting actions remain available to resolve missing information or temporary conflicts.

B.3.4 Asynchronous Planning and Commit-Time Coordination

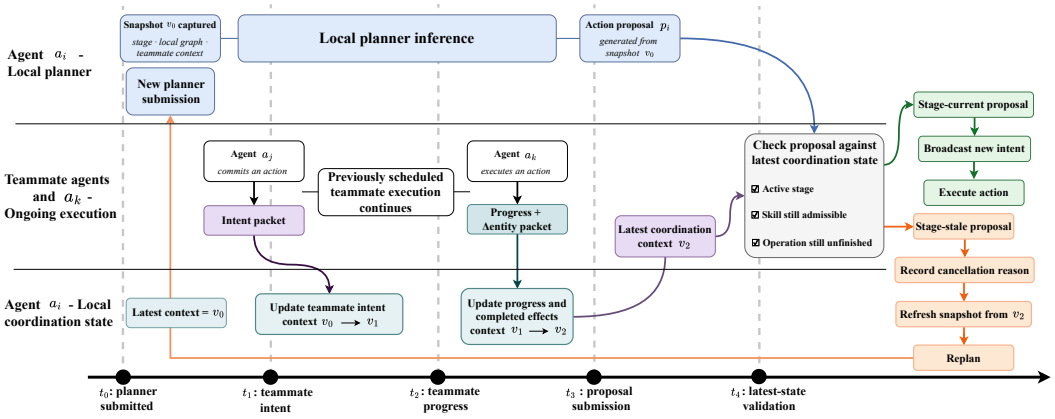

Figure 6: Asynchronous planning with commit-time validation. An agent generates an action proposal from a local snapshot while teammate agents continue concurrent execution. Incoming intent and progress updates refresh the coordination state, and each returned proposal is validated against the latest state before execution. Stale proposals are regenerated from updated context.

Figure 6 shows the lifecycle of an asynchronous planning request and its commit-time validation. At t_0, agent a_i captures a local snapshot v_0, including its current stage, local world graph, and latest teammate context. The local planner performs inference on this snapshot and generates an action proposal p_i, while other agents continue execution concurrently.

At t_1 and t_2, teammate agents broadcast incremental intent and progress packets, updating the local coordination state from v_0 to v_1 and v_2. These updates do not interrupt the ongoing planner inference. When the proposal is submitted at t_3, it is instead validated against the latest coordination state v_2, rather than the outdated snapshot v_0.

Commit-time validation checks whether the corresponding stage remains active, the proposed skill is still admissible under the updated coordination context, and the operation remains unfinished. A valid proposal is committed to the executor and its intent is broadcast incrementally. Otherwise, the proposal is deferred, the reason is recorded in the agent's refreshed context, and replanning is triggered.

For compactness, let $\mathcal{S} = (S_1, \ldots, S_K)$ denote the execution stages obtained from instruction I. We write $\mathcal{U}_i^t \subseteq \mathcal{U}_i$ for the skills currently admissible to agent i after grounding, prerequisite, ownership, and conflict checks. A returned proposal is represented as $p_i^t = (u_i, \theta_i, z_i)$, where u_i is the proposed skill, θ_i its grounded arguments, and z_i the stage version from which the proposal was generated.

We use

$$\mathrm{Valid}(p_i^t) = \mathbf{1}\left[z_i = z \ \wedge \ u_i \in \mathcal{U}_i^t \ \wedge \ \mathrm{op}(p_i^t) \notin L_z\right]$$

to denote commit-time validity. We write $\mathrm{StageFinished}(z) = 1$ when the LLM judges stage z complete from policy-visible observations and execution feedback, or when the runtime skips it after repeated unsuccessful attempts or prolonged execution, provided that no state-changing skill of that stage remains in flight. This denotes a control-flow decision; unmet requirements remain unsatisfied when a stage is skipped.

Algorithm 1 Event-driven decentralized execution with validated commits

Require: instruction I, agents $\mathcal{A} = \{a_i\}_{i=1}^{N}$, local states $\{G_i, M_i, Q_i, f_i\}_{i=1}^{N}$

$\mathcal{S} = (S_1, \ldots, S_K) \leftarrow \textsc{Decompose}(I)$, $z \leftarrow 1$, active $\leftarrow 1$
$\mathcal{C}_z \leftarrow \textsc{Contract}(S_z)$, $L_z \leftarrow \varnothing$
$i_z^\star \leftarrow \textsc{FirstReadyAgent}(\mathcal{A})$
$\mathcal{W}_z \leftarrow \mathrm{TW}(M_{i_z^\star})$
$\mathcal{H}_z \leftarrow \mathrm{TEG}(\mathcal{C}_z, G_{i_z^\star}, \mathcal{W}_z)$
$\mathcal{C}_z \leftarrow \textsc{Refine}(\mathcal{C}_z, \mathcal{H}_z)$
while active $= 1$ **do**
  $\mathcal{E}^t \leftarrow \textsc{CollectEvents}()$
  $(G_i^t, f_i^t, \mathrm{Idle}_i^t) \leftarrow \textsc{Assimilate}(\mathcal{E}^t, G_i^{t-1}, f_i^{t-1})$, $\forall i$
  $Q_i^t \leftarrow \textsc{ValidatePackets}(Q_i^{t-1})$, $\forall i$; $\textsc{Broadcast}(\Delta_{\mathrm{prog}}^t, \Delta_{\mathrm{ent}}^t)$
  $L_z \leftarrow \textsc{Refresh}(\mathcal{C}_z, L_z, \{G_i^t, Q_i^t, f_i^t\})$
  if $\textsc{StageFinished}(z)$ **then**
    if $z = K$ **then**
      active $\leftarrow 0$
    else
      $z \leftarrow z + 1$
      $\mathcal{C}_z \leftarrow \textsc{Contract}(S_z)$, $L_z \leftarrow \varnothing$
      $i_z^\star \leftarrow \textsc{FirstReadyAgent}(\mathcal{A})$
      $\mathcal{W}_z \leftarrow \mathrm{TW}(M_{i_z^\star})$
      $\mathcal{H}_z \leftarrow \mathrm{TEG}(\mathcal{C}_z, G_{i_z^\star}, \mathcal{W}_z)$
      $\mathcal{C}_z \leftarrow \textsc{Refine}(\mathcal{C}_z, \mathcal{H}_z)$
    end if
  end if
  if active $= 1$ **then**
    $\mathcal{U}_i^t \leftarrow \textsc{Filter}(\mathcal{U}_i, G_i^t, Q_i^t, L_z)$, $\forall i$
    $\mathcal{R}^t \leftarrow \varnothing$
    for all returned $p_i^t = (u_i, \theta_i, z_i)$ **do**
      if $z_i = z \wedge u_i \in \mathcal{U}_i^t \wedge op(p_i^t) \notin L_z$ **then**
        $\textsc{Broadcast}(\textsc{Intent}(p_i^t))$, $\textsc{Execute}(p_i^t)$
      else
        $\mathcal{R}^t \leftarrow \mathcal{R}^t \cup \{a_i\}$
      end if
    end for
    for all idle or replanning a_i requiring a decision **do**
      $\textsc{Submit}[\pi_i(\mathcal{C}_z, G_i^t, Q_i^t, f_i^t, \mathcal{U}_i^t)]$
    end for
  end if
end while
$m_i^{\mathrm{new}} \leftarrow \textsc{FinalizeEpisode}(a_i)$, $M_j \leftarrow M_j \cup \{m_i^{\mathrm{new}}\}$, $\forall i, \forall j \neq i$

Algorithm 1 summarizes the event-driven runtime. Rather than enumerating low-level message-handling steps, the algorithm groups them into four state transitions: event assimilation, execution-state refresh, commit-time validation, and planner submission. The corresponding communication and recovery rules are described in Sections B.3 and B.4.

B.4 Feedback-Driven Recovery and Progress Preservation

Executor responses close the control loop between semantic skill selection and environmental affordances. The attempted skill, success status, and structured failure reason update the acting agent's local state and determine the appropriate recovery action. Recovery follows a bounded repair process that preserves the task specified by the instruction. Table 8 summarizes the main execution failures and their corresponding recovery behavior.

Recovery is driven directly by executor feedback and aims to repair an operation without changing the task specified by the instruction. When required grounding is missing, the agent first gathers

Table 8: Recovery behavior for common execution failures. Recovery first attempts a local repair while preserving the task specified by the instruction; failures that cannot be repaired remain explicit.

Execution problem	Recovery action	What is kept unchanged
Required object or state is not sufficiently observed	Gather more evidence through navigation, exploration, or a relevant teammate update before acting.	Do not guess an unsupported grounding.
A required physical precondition is not yet satisfied	Establish the missing precondition, or wait for the agent currently responsible for it.	Keep the original operation in the task.
The executor rejects the requested spatial relation	Try at most one other legal realization that satisfies the same task constraint.	Keep the requested object, destination, and spatial requirement.
A teammate has already completed the operation	Discard the stale proposal and replan from the updated team state.	Do not repeat an already completed effect.
No supported realization succeeds	Mark the operation as unresolved and allow unrelated operations within the active stage to continue.	Keep the failure explicit rather than treating the operation as completed.
Repeated attempts or prolonged execution fail to meet stage requirements	By incorporating the failure count and reasons for repeated failures into the prompt, the LLM is biased toward skipping the current stage and proceeding to the next one; if the current stage is the final one, the episode is terminated.	Preserve unmet requirements rather than marking the skipped stage as successfully completed.

additional observations; when a physical precondition is unmet, it establishes that precondition; and when the executor reports an incompatible relation, it may try a bounded legal alternative. Throughout recovery, the requested entity, destination, state, and spatial constraints remain unchanged. If no supported repair succeeds, the operation remains explicitly unresolved rather than being treated as completed.

A failure will be communicated to teammates. When a failure impacts task progress in the current activity stage, dependent and conflicting operations are flagged, deprioritized, and deferred to subsequent replanning. If repeated attempts or prolonged execution fail to yield success, information such as the number of executions and the causes of failure will be leveraged as references for subsequent feedback-guided replanning.

C Additional Experiments

The main experiments evaluate the overall effectiveness of our framework, the contributions of TW and TEG, robustness under historical and execution interference, and performance on episodes exhibiting different continuity requirements. This section provides complementary experiments and analyses that further characterize the framework from several aspects. We first compare TW–TEG with alternative embodied-memory mechanisms under the same multi-agent collaboration pipeline. We then examine multi-agent memory collaboration through single- versus multi-agent execution, cross-agent information sharing, and increasing team size. Next, we analyze the internal retrieval-to-binding process and execution and planning efficiency across different LLM backbones. We further evaluate the stability of the results under semantics-preserving prompt reformulations and independently sampled evaluation suites. Finally, we present a representative failure case and illustrate remaining limitations.

C.1 Comparison with Alternative Memory Mechanisms

We conduct a controlled comparison with alternative embodied-memory mechanisms. Directly comparing against the original single-agent systems would confound memory design with differences in communication, task allocation, planning, and execution. We therefore adapt the memory mechanisms of 3DSPMR, FindingDory, and MEMENTO to our multi-agent framework and replace only TW–TEG, while retaining the same inter-agent communication, shared-memory exchange, planning, execution, and evaluation pipeline. All variants are evaluated on matched episodes with the same DeepSeek-V4 and Qwen3-8B backbones. This design isolates how different ways of organizing and using historical experience affect downstream collaborative execution.

Table 9: Comparison with alternative memory mechanisms adapted to the same multi-agent collaboration framework. TW–TEG is replaced by memory mechanisms adapted from 3DSPMR, FindingDory, and MEMENTO, while all non-memory modules use our unchanged collaboration pipeline. Results are reported with matched DeepSeek-V4 and Qwen3-8B backbones.

Method	**Backbone**	**Task performance**				**Execution and planning**		**Impossible-task detection**	
		TC (%) ↑	**SR** (%) ↑	**ΔTC** (pp) ↑	**ΔSR** (pp) ↑	**Steps** ↓	**Avg. replans** ↓	**Exact Hit** (%) ↑	**Coarse Recall** (%) ↑
3DSPMR (Cai et al., 2025)	DeepSeek-V4	28.71	3.67	-27.28	-15.33	9425.52	45.42	0.00	0.00
	Qwen3-8B	14.32	0.33	-26.58	-9.67	9088.09	45.66	0.00	0.00
FindingDory (Yadav et al., 2025)	DeepSeek-V4	48.66	15.33	-7.33	-3.67	13239.36	39.11	0.00	0.00
	Qwen3-8B	21.44	1.33	-19.46	-8.67	9444.34	44.54	0.00	0.00
MEMENTO (Kwon et al., 2025)	DeepSeek-V4	51.33	16.67	-4.66	-2.33	13869.52	40.77	0.00	0.00
	Qwen3-8B	21.75	2.00	-19.15	-8.00	**8874.10**	45.99	0.00	0.00
Ours	DeepSeek-V4	**55.99**	**19.00**	–	–	10876.40	**18.45**	**14.58**	**73.91**
	Qwen3-8B	40.90	10.00	–	–	13808.72	26.21	4.17	45.65

As shown in Table 9, TW–TEG consistently outperforms the adapted memory mechanisms in task completion and strict success under both backbones. The difference is much larger with Qwen3-8B, suggesting that structured organization and grounding of distributed history become more important when the backbone is less capable of reconstructing historical dependencies on its own.

The differences are also reflected in execution behavior. TW–TEG requires substantially fewer replans than the alternatives while maintaining higher task progress, indicating more stable conversion of historical evidence into executable decisions. More importantly, it is the only method that achieves non-zero impossible-task detection, whereas all adapted alternatives fail on both fine- and coarse-grained detection. Together, these results show that the gain does not arise merely from retaining historical information, but from how that information is organized, grounded, and used during collaborative execution.

C.2 Analysis of Multi-Agent Memory Collaboration

We analyze multi-agent memory collaboration through three experiments. First, we compare single-agent and two-agent execution to examine the benefit of collaboration under matched memory and reasoning mechanisms. Second, we remove the cross-agent sharing mechanism to distinguish the effect of shared information from parallel execution alone. Finally, we increase the team size to study how task performance and coordination costs change as collaboration scales.

C.2.1 Single-Agent versus Two-Agent Execution

We first examine whether multi-agent execution provides benefits beyond those obtainable by a single embodied agent equipped with the same memory and reasoning mechanisms. We compare the single-agent and two-agent variants on the same evaluation set. We evaluate the difference from two complementary perspectives. Table 10 measures how far tasks progress, both at termination and under fixed resource budgets, while Fig. 7 further evaluates task completion performance under increasing simulation-step and planning-call budgets.

We use task completion (TC) as the common progress measure in Table 10. TC@B steps denotes the mean TC available at simulator-step budget B, and TC@K plans denotes the mean TC when

Table 10: Task completion and fixed-resource TC efficiency for single-agent and two-agent execution over the same evaluation set. Gap denotes two-agent minus single-agent performance, and Single-agent drop measures the relative reduction from the two-agent result.

Measure	Multi-agent	Single-agent	Gap	Single-agent drop	95% CI for gap
Percent-complete outcomes					
Mean TC	**55.99%**	51.26%	+4.74 pp	8.46%	$[+0.37, +9.48]$ pp
TC ≥ 0.50	**62.67%**	53.33%	+9.33 pp	14.89%	$[+2.67, +16.33]$ pp
TC ≥ 0.60	**51.00%**	40.00%	+11.00 pp	21.57%	$[+4.00, +18.33]$ pp
TC ≥ 0.70	**41.00%**	30.33%	+10.67 pp	26.02%	$[+2.33, +19.67]$ pp
TC ≥ 0.80	**29.33%**	20.33%	+9.00 pp	30.69%	$[+2.33, +16.33]$ pp
TC ≥ 0.90	**23.67%**	14.67%	+9.00 pp	38.02%	$[+3.33, +15.00]$ pp
TC over simulation-step budgets					
TC@5k steps	**46.96%**	41.57%	+5.39 pp	11.48%	$[+1.45, +9.79]$ pp
TC@10k steps	**54.12%**	49.48%	+4.64 pp	8.58%	$[-0.48, +9.13]$ pp
TC over total planning-call budgets					
TC@10 plans	**41.98%**	31.42%	+10.56 pp	25.15%	$[+5.07, +16.71]$ pp
TC@20 plans	**49.96%**	42.22%	+7.74 pp	15.49%	$[+3.58, +12.22]$ pp
TC@50 plans	**55.01%**	51.13%	+3.88 pp	7.05%	$[+0.50, +8.62]$ pp
TC@100 plans	**55.99%**	51.26%	+4.74 pp	8.46%	$[+0.34, +9.49]$ pp

the cumulative number of completed high-level plans reaches K. For two-agent execution, planning calls from both agents are summed, so the multi-agent system does not receive an additional planning allowance.

Because episodes within the same dataset belong to the same continuous task stream, we conduct statistical analysis at the dataset level on the same evaluation set. The 95% confidence interval (CI) gives the uncertainty range of the two-agent minus single-agent performance gap; a CI above zero indicates a consistently positive advantage.

Table 10 shows that the benefit of two-agent execution becomes more pronounced as the completion criterion becomes stricter. The gap is modest in mean TC but grows consistently at higher completion thresholds, indicating that collaboration mainly helps push partially completed episodes toward near-complete outcomes rather than providing a uniform increase in progress.

The same pattern appears under fixed resource budgets. Two-agent execution maintains higher TC with the same simulation-step and planning budgets, with the largest advantage occurring when planning opportunities are limited and the gap narrowing as more plans become available. This suggests that collaboration improves how effectively limited planning resources are converted into task progress, rather than benefiting simply from additional execution or LLM calls.

Figure 7 complements the TC analysis with the stricter criterion of complete episode Success Rate (SR). Across the entire simulation-step range, the two-agent curve remains above the single-agent curve. Both configurations benefit from additional embodied interaction, but the collaborative system reaches successful execution earlier and preserves its advantage as the budget grows. The 8.00-pp gap highlighted at 10k steps shows that the higher intermediate TC observed in the table is not merely partial progress; it also translates into a greater probability of fully completing an episode within the same execution budget.

The planning-efficiency curve exhibits an even clearer separation. The two-agent system gains strict success rapidly with the first few high-level plans, whereas the single-agent system improves more slowly. The largest separation occurs in the low-budget region, including the 12.00-pp gap at 20 aggregate plans, after which both curves gradually approach their respective plateaus. This behavior closely matches the TC results: collaboration is most valuable when planning opportunities are scarce, while additional planning resources allow the single-agent system to recover part, but not all, of the gap.

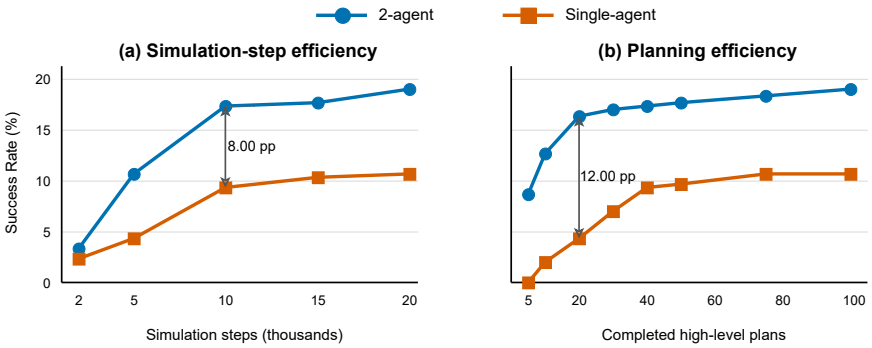

Figure 7: Strict-success comparison between two-agent and single-agent execution under increasing resource budgets. (a) Success rate as the available simulation-step budget increases. (b) Success rate as the aggregate number of completed high-level plans increases, with calls from both agents summed in the two-agent setting. The annotated gaps highlight the two-agent advantage at representative intermediate budgets.

C.2.2 Effect of Shared Cross-Agent Information

The preceding single- versus two-agent comparison shows that collaborative execution improves both task progress and resource-constrained performance. We further examine whether these gains depend on information accumulated and shared across agents, rather than merely on parallel embodied execution. Specifically, we compare the full two-agent framework with a *No-share* variant in which each agent retains only its own historical information and receives no cross-agent shared information. Communication and historical sharing are coupled in our framework, since exchanged information can become retained experience. The agents, task setting, backbone, and execution protocol otherwise remain unchanged.

We evaluate three complementary aspects: outcome quality, execution cost, and coordination behavior. For outcomes, we report success rate, mean task completion, the median, and lower completion percentiles to characterize the entire completion distribution. For efficiency, we measure simulator steps, budget saturation, and replanning frequency, together with runtime, simulator steps, and LLM plans normalized by achieved task completion. One *completion-equivalent* corresponds to one unit of accumulated normalized task completion, allowing resource costs to be compared at an equivalent level of useful progress. Coordination is characterized by executed high-level actions, Wait-related behavior, task-action share, and the invalid-before-execution rate.

As shown in Table 11, removing the sharing mechanism degrades task completion while increasing execution and planning costs. The *No-share* variant performs more high-level actions but spends a larger fraction of them on waiting and fewer on task-directed behavior, indicating that the absence of cross-agent information leads to less efficient coordination rather than simply less activity.

More importantly, the removed mechanism affects not only immediate information exchange but also shared memory: information obtained by one agent can no longer be retained and reused by others in later execution. As a result, teammate discoveries, completed operations, and historical state changes must be rediscovered or inferred independently, weakening both current coordination and cross-episode reuse of distributed experience. The results therefore show that sharing contributes through both online coordination and the accumulation of reusable cross-agent history. And these results distinguish our setting from single-agent memory: performance depends not only on retaining history, but on accumulating and reusing experience distributed across teammates.

C.2.3 Scaling Collaboration from Two to Six Agents

We further examine how memory-continuous multi-agent collaboration changes as more agents participate in the team. The evaluation suite, memory mechanism, and execution protocol remain fixed, while only the configured team size is varied. We select three representative backbones with different reasoning capabilities and analyze scaling from three complementary perspectives: task performance and execution cost, communication behavior, and agent-level contribution.

Table 11: Effect of removing cross-agent shared information. The No-share variant restricts each agent to its own history, while Shared denotes the full framework with cross-agent information sharing. Results compare outcome quality, execution cost, and coordination behavior. The Gap column reports the degradation or additional cost introduced by removing shared information; arrows indicate the preferred direction of each metric. For Replans / episode, replanning events are summed across all agents within each episode and then averaged over episodes.

Dimension	Metric	Desired	No-share	Shared	Gap
Outcome quality	Success rate	↑	15.33%	**19.00%**	−3.67 pp
	Mean task completion	↑	48.41%	**55.99%**	−7.58 pp
	Median task completion	↑	45.64%	**60.00%**	−14.36 pp
	25th-percentile completion	↑	14.29%	**25.00%**	−10.71 pp
	10th-percentile completion	↑	2.78%	**6.56%**	−3.78 pp
Efficiency / cost	Mean simulator steps / episode	↓	14094.93	**10876.40**	+29.59%
	Step-budget saturation (≥ 19,990)	↓	52.33%	**31.00%**	+21.33 pp
	Replans / episode	↓	41.20	**36.89**	+11.69%
	Runtime / completion-equivalent	↓	4185.79 s	**3619.20 s**	+15.66%
	Simulator steps / completion-equivalent	↓	29113.06	**19425.18**	+49.87%
	LLM plans / completion-equivalent	↓	84.72	**65.66**	+29.03%
Coordination behavior	Executed high-level actions / episode	↓	46.23	**27.22**	+69.85%
	Wait share among executed actions	↓	43.42%	**21.93%**	+21.49 pp
	Wait actions / completion-equivalent	↓	41.46	**10.66**	+288.86%
	Task-action share among executed actions	↑	28.39%	**33.79%**	−5.40 pp
	Invalid-before-execution attempt rate	↓	2.14%	**1.40%**	+0.74 pp

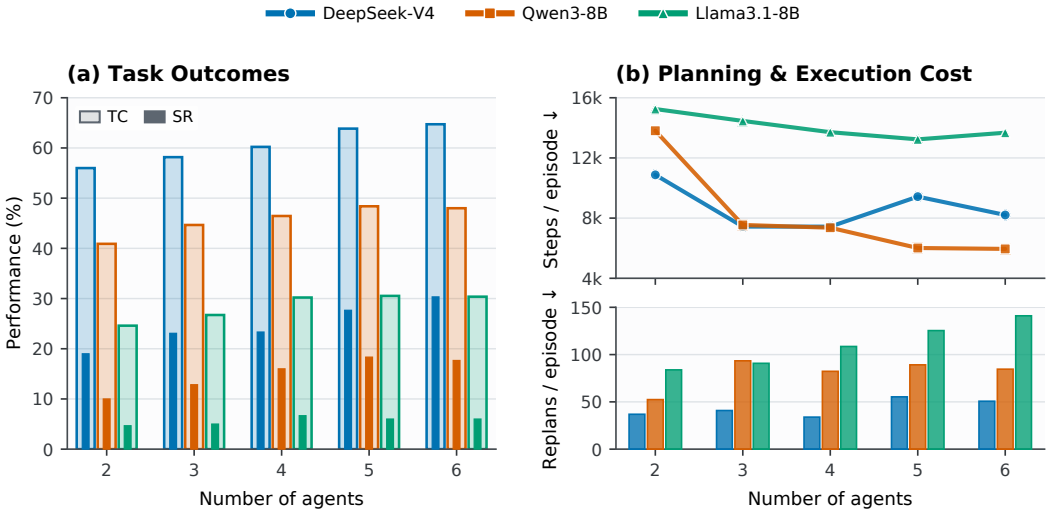

Figure 8: Task performance and execution behavior under increasing team size across three representative LLM backbones. TC and SR characterize task outcomes, while execution steps and replanning characterize embodied-execution and planning cost. Replanning events are summed across all agents within each episode.

Task performance and execution cost. Figure 8 summarizes task performance and execution behavior as the team grows.

Increasing the team size generally improves task completion, but the extent to which additional agents are converted into useful progress depends strongly on the backbone. DeepSeek-V4 continues to improve in both TC and SR throughout the tested range, indicating that additional parallel capacity remains useful even for larger teams. Qwen3-8B exhibits a different regime: performance improves consistently at first but eventually saturates, with further reduction in execution steps no longer accompanied by a corresponding improvement in task outcome. Llama3.1-8B reaches this regime

Table 12: Communication cost and message quality under increasing team size. Msg(/Ep) counts directed messages per episode. Updates(/Agent/Ep) removes recipient-side broadcast fan-out and is computed as $\mathrm{Msg}/\mathrm{Ep}/(N(N-1))$. Words(/Ep) measures transmitted text volume. Trunc. and Failure report message truncation and communication failure, while Applied denotes the fraction of received messages that are applied to the receiver's coordination state.

Backbone	Number	Msg	Updates	Words	Trunc. (%)	Failure (%)	Applied (%)
DeepSeek-V4	2	78.29	39.15	34,515	38.91	4.45	62.43
	3	201.30	33.55	111,459	58.55	3.23	54.27
	4	373.06	31.09	203,340	57.24	4.56	59.53
	5	619.49	30.97	331,298	54.92	4.95	53.92
	6	825.33	27.51	431,252	55.50	3.60	53.20
Qwen3-8B	2	86.27	43.14	36,056	33.75	12.76	48.16
	3	180.07	30.01	75,376	31.44	11.04	48.49
	4	269.27	22.44	125,861	41.24	9.08	50.76
	5	373.08	18.65	177,589	44.84	7.20	61.60
	6	487.97	16.27	241,773	48.95	5.63	66.22
Llama3.1-8B	2	60.99	30.50	22,502	22.17	22.10	36.28
	3	145.66	24.28	59,741	29.26	9.20	52.76
	4	252.62	21.05	113,013	38.74	10.78	50.45
	5	406.72	20.34	182,660	38.73	13.55	50.77
	6	531.68	17.72	239,267	41.50	11.14	53.49

earlier; additional agents improve partial completion to some extent, while strict success changes only modestly.

The execution statistics reveal why larger teams do not yield proportional performance gains. More agents generally reduce the embodied trajectory needed to make progress, consistent with increased parallel exploration and execution. However, the aggregate replanning burden does not decrease accordingly and can increase substantially as the team becomes larger. This produces two opposing effects: additional agents provide more opportunities for concurrent progress, but they also enlarge the set of simultaneously evolving plans and execution states that must remain mutually consistent. Stronger backbones continue to convert the additional parallel capacity into task gains, whereas weaker backbones enter a coordination-limited regime earlier, where additional parallelism yields diminishing returns.

Communication scaling. We next examine whether the performance gains from larger teams are accompanied by corresponding growth in communication overhead. Because each semantic coordination update is transmitted separately to every other configured agent, the number of directed messages mechanically grows with the number of recipients. To separate this broadcast fan-out from the frequency with which agents actually generate new coordination information, we additionally report Updates/Agent/Ep, defined as $\mathrm{Msg}/\mathrm{Ep}/(N(N-1))$.

Table 12 shows that aggregate communication cost grows much faster than task performance as the team expands. Directed messages and transmitted words increase sharply for all three backbones. This growth is partly structural: each new coordination update must be delivered separately to more teammates as the team becomes larger. After removing this recipient multiplicity, however, Updates/Agent/Ep consistently decreases with team size. The rapid increase in aggregate traffic therefore arises primarily from wider information dissemination rather than from each individual agent generating coordination updates more frequently.

The message-level diagnostics further show that larger traffic volume does not simply translate into communication failure. Failure rates do not increase monotonically with team size, and a substantial fraction of received messages continues to be incorporated into the receiver's coordination state. The more consistent scaling effect is increased truncation pressure, particularly for Qwen3-8B and Llama3.1-8B, indicating that the bounded communication payload must accommodate information

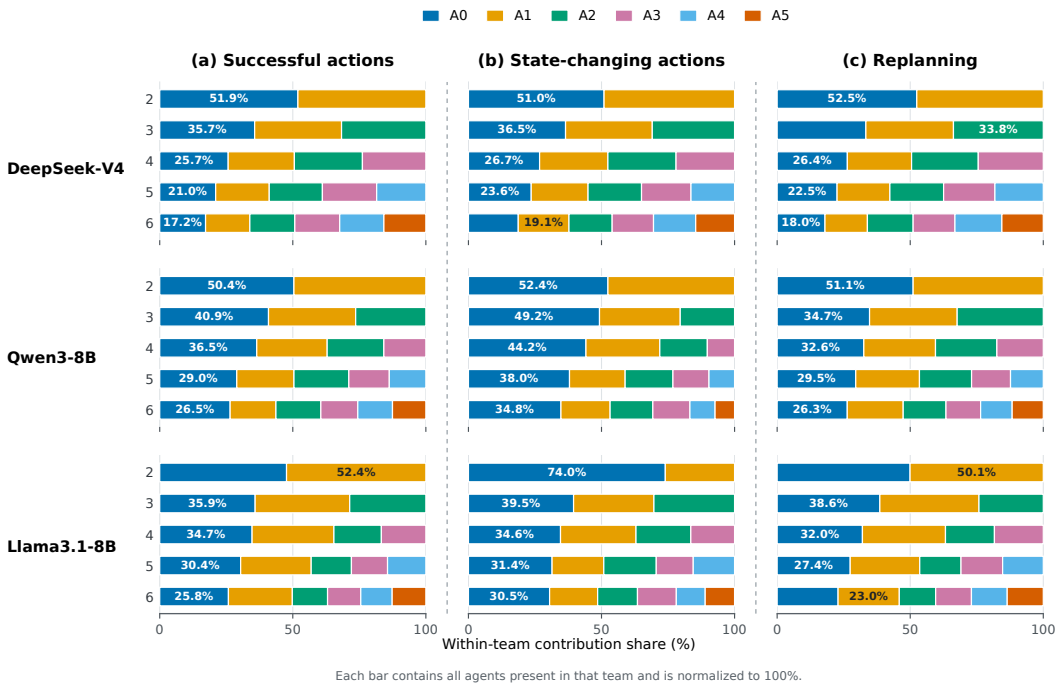

Figure 9: Agent-level contribution under increasing team size across three representative LLM backbones. Successful Actions measures completed non-atomic task actions, State-changing reports each agent's fraction of successful state-changing actions, and Replans characterizes the agent-level planning burden.

from a larger set of concurrently active agents. At the same time, the applied-message rate increases substantially for these two backbones, showing that larger teams do provide additional coordination information that can be used by the receiver. Nevertheless, the performance saturation observed in Figure 8 indicates that increasing the amount of useful shared information is not sufficient by itself: once planning and execution coordination become the dominant bottleneck, additional communication no longer translates proportionally into task completion.

Agent-level contribution. Finally, we examine whether the additional agents introduced by team scaling contribute comparably to productive task execution.

Figure 9 reveals a clear difference in how effectively larger teams utilize their additional members. DeepSeek-V4 maintains a comparatively balanced distribution of successful actions, state-changing contributions, and replanning activity as the team grows. Additional agents therefore continue to participate in productive execution, which is consistent with the continued improvement in TC and SR observed in Figure 8.

The weaker backbones exhibit a different pattern. For Qwen3-8B, successful state-changing contributions become increasingly concentrated among a subset of agents as the team grows, and the replanning workload shows a similar asymmetry. Llama3.1-8B also exhibits substantial imbalance in larger teams, although the exact pattern is less monotonic. Importantly, successful-action counts can remain considerably less skewed than State Share. Thus, agents may remain active and execute valid actions without contributing a proportional amount of useful state change.

This imbalance helps explain why planning and communication cost can continue to grow after task performance begins to saturate. Marginally added agents do not consistently absorb enough productive state-changing work to make the team contribution more balanced. They nevertheless participate in concurrent planning and execution, enlarging the number of simultaneously active proposals, object interactions, and execution states that must be reconciled. As the team grows, this creates more opportunities for overlapping operations, outdated proposals, and inconsistent local execution contexts, which in turn require additional coordination, replanning, or execution correction.

The marginal agent can therefore introduce substantial planning and execution overhead without producing a comparable amount of additional task progress.

Taken together, these results show that multi-agent scaling is governed by a trade-off between parallel execution capacity and coordination efficiency. Larger teams can provide broader parallel coverage and more distributed information, but they also amplify communication fan-out and the interactions among concurrent planning and execution processes. When additional agents continue to perform productive and complementary state-changing work, the extra capacity can translate into improved task performance. When marginal agents contribute progressively less useful task change while still participating in planning and execution, coordination cost grows faster than task progress, producing diminishing returns from further team scaling.

C.3 Mechanistic Diagnostics and Efficiency Analysis

The main experiments show that reliable memory utilization depends on both structured historical evidence and its conversion into executable decisions. Here, we further examine *where* errors arise within this process and *how efficiently* different backbones convert memory-grounded reasoning into task progress. We first complement the main-text retrieval and grounding metrics with diagnostics that separate historical-dependency triggering, request quality, candidate discrimination, and end-to-end binding. We then analyze execution throughput and planning overhead across backbones to determine whether stronger task performance is accompanied by more effective use of simulation, runtime, and replanning resources. Together, these analyses connect the internal quality of historical grounding to its downstream execution cost.

C.3.1 Fine-Grained Historical Retrieval and Binding Diagnostics

The main-text results already measure whether required historical information is requested, whether the correct entities are retrieved, and whether the final grounding is exact. Rather than repeating these aggregate measures, we introduce complementary diagnostics that separate three stages that can fail after an instruction encounters a historical dependency: deciding whether retrieval should be invoked, constructing a productive retrieval request, and selecting the historically consistent interpretation from competing candidates.

Table 13: Fine-grained diagnostics of historical-dependency triggering, retrieval request quality, and candidate selection across LLM backbones. *Gate precision* is the fraction of all historical retrievals triggered during execution that correspond to genuinely required historical dependencies; unlike the main-text retrieval metric, its denominator is triggered retrievals rather than retrievals required by the task. *Distinct-target yield* is the ratio of distinct verified historical targets recovered to unique retrieval requests; repeated recovery of an already retrieved target does not increase the numerator, so redundant or unproductive requests reduce the score. *Ambiguous-candidate exact selection* evaluates entity-level selection only when the complete verified target set has already been retrieved together with competing candidates. Its denominator is the number of retrieved entities, whereas the corresponding main-text metric is retrieval-level, includes retrievals regardless of whether the complete target set is recovered, and uses retrieval attempts as its denominator.

	Trigger and request quality		Selection robustness
LLM backbone	Gate precision ↑	Distinct-target yield ↑	Ambiguous-candidate exact selection ↑
GPT-5.5	**98.4%**	<u>55.5%</u>	**63.5%**
DeepSeek-V4	<u>98.1%</u>	**56.1%**	<u>52.5%</u>
Qwen3-32B	97.2%	37.6%	46.6%
Qwen3-8B	97.1%	41.0%	42.1%
Llama-4-Scout	90.3%	30.6%	51.9%
Llama-3.3-70B	96.9%	37.9%	52.4%
Llama-3.1-8B	86.5%	29.6%	40.9%

Table 14: End-to-end exact historical binding across different dependency structures. Results are separated by the number of entities that must be bound jointly and by whether the historical argument specifies an object or a location. Exact binding requires a correct retrieval request, recovery of the complete verified target set, and an exactly correct final binding.

LLM backbone	Target-set cardinality		Historical argument role	
	Single-entity E2E exact ↑	Multi-entity E2E exact ↑	Object E2E exact ↑	Location E2E exact ↑
GPT-5.5	12.4%	**7.3%**	11.8%	12.8%
DeepSeek-V4	**15.0%**	2.7%	**13.9%**	**14.9%**
Qwen3-32B	10.9%	3.6%	10.7%	10.1%
Qwen3-8B	10.7%	2.7%	11.3%	7.9%
Llama-4-Scout	6.5%	0.9%	5.1%	8.1%
Llama-3.3-70B	5.5%	4.5%	5.2%	5.8%
Llama-3.1-8B	4.4%	2.7%	4.5%	4.1%

Table 13 shows that historical-dependency triggering is generally much more reliable than the subsequent grounding process. Most backbones achieve high Gate precision, indicating that they rarely invoke historical retrieval without a valid reason. The larger differences emerge after this decision has been made. Distinct-target yield is substantially lower and varies more strongly across models, showing that recognizing the need for history does not guarantee that the resulting request will recover a sufficiently informative and non-redundant set of historical targets.

Candidate selection introduces a second bottleneck. Even when the complete verified target set is already present among the retrieved candidates, adding plausible alternatives causes a substantial loss in exact binding. The strongest models also exhibit different strengths within this pipeline. DeepSeek shows the strongest request yield, whereas GPT-5.5 is substantially stronger in exact selection under ambiguity. This is consistent with the main-text observation that broader historical retrieval does not necessarily translate into more reliable final grounding.

We next examine whether exact binding difficulty depends primarily on the number of historical targets that must be resolved or on their semantic role. Table 14 reports end-to-end exact binding under these two complementary partitions. An end-to-end binding is counted as exact only when the required historical dependency is correctly requested, the complete verified target set is recovered, and the final prediction exactly matches that set. The cardinality partition distinguishes dependencies requiring one entity from those requiring multiple entities simultaneously. The semantic-role partition separately distinguishes historical object arguments from historical source or target locations. These partitions are computed independently and may overlap; for example, a single-entity dependency may also correspond to an object argument. A small number of layout/configuration and uncategorized dependencies are omitted from the object/location comparison because they do not provide sufficient support for a stable role-specific analysis.

The cardinality comparison exposes the clearest structural difficulty. End-to-end exact binding drops markedly when a historical dependency requires multiple entities to be resolved jointly. This pattern appears broadly across backbones and is more informative than the exact ranking within the multi-entity subset. Resolving one historical entity already requires the retrieval and selection stages to agree on a consistent interpretation; multi-entity dependencies additionally require several bindings to remain mutually compatible with the same historical context. An error in any one entity is sufficient to invalidate the complete result. The sharp degradation therefore identifies compositional consistency across multiple historical entities as a major remaining bottleneck.

By contrast, the object/location partition does not reveal a similarly universal asymmetry. Individual backbones vary in which role they handle more reliably, but the two categories remain broadly comparable relative to the much larger single- versus multi-entity gap. This suggests that current grounding difficulty is determined less by whether an operation requires a historical object or location and more by whether multiple pieces of historical evidence must be bound coherently at the same time.

Table 15: Execution throughput and planning overhead across LLM backbones. Throughput metrics normalize achieved task completion by simulator steps, runtime, and replanning, while planning-overhead metrics report historical memory-update latency and plan-time historical regrounding.

Backbone	Execution throughput			Planning overhead	
	TC/1K steps ↑	TC/runtime min. ↑	TC/replan ↑	Hist.-Upd. (s) ↓	Reground ↓
GPT-5.5	**6.5892**	**1.7909**	**1.6713**	0.3656	**40.29**
DeepSeek-V4	5.1480	1.6578	1.5178	**0.3441**	44.27
Qwen3-32B	5.0113	1.6374	0.7064	0.4492	67.92
Llama-4-Scout	3.9008	1.0941	0.4343	0.5593	77.25
Qwen3-8B	2.9620	0.3693	0.7805	0.4753	51.67
Llama-3.3-70B	3.3627	0.9341	0.4697	0.7143	78.11
Llama-3.1-8B	3.2260	0.2700	0.2934	0.8208	82.49

Together, Tables 13 and 14 refine the failure pattern observed in the main experiments. The system is generally capable of recognizing when historical reasoning is required, but reliability decreases as information must be converted from a retrieval request into productive evidence, from retrieved evidence into an unambiguous binding, and finally from individual bindings into a mutually consistent multi-entity interpretation. The dominant challenge is therefore not simply "finding history," but preserving historical consistency throughout the complete retrieval-to-binding process.

C.3.2 Execution Throughput and Planning Overhead

We next examine whether the performance differences across backbones are accompanied by corresponding differences in resource efficiency. We normalize achieved task completion by several forms of execution and planning cost. TC/1K steps measures the amount of task completion obtained per thousand simulator steps, TC/runtime min. measures task completion obtained per minute of runtime, and TC/replan measures completion obtained per replanning event.

Table 15 shows that stronger backbones achieve higher execution throughput rather than simply consuming more resources. GPT-5.5 performs best across all three normalized efficiency metrics, with DeepSeek-V4 consistently close behind, indicating that stronger models convert simulation steps, runtime, and replanning into task progress more effectively.

Planning overhead shows a similar pattern. DeepSeek-V4 has the lowest history-update latency, while GPT-5.5 requires the fewest regroundings. Although the ordering is not identical across metrics, lower-throughput models generally require more corrective grounding. This suggests that execution efficiency depends on how reliably the backbone interacts with the memory–planning pipeline, rather than on model scale alone.

C.4 Stability and Evaluation Validity

Beyond method-level effectiveness, we examine whether the conclusions reported in the main experiments are sensitive to incidental choices in the evaluation procedure. We consider two complementary sources of variation. First, we test whether the framework depends on a particular prompt realization by reformulating prompt wording, organization, and structure while preserving the underlying task semantics. Second, we evaluate independently sampled difficulty-balanced subsets to determine whether the reported performance depends on the particular 300 episodes used in the main evaluation. Together, these experiments test whether the observed conclusions remain stable against prompt formulation and evaluation-set sampling.

C.4.1 Robustness to Semantics-Preserving Prompt Reformulations

LLM-based embodied systems can be sensitive to prompt design, raising the possibility that an observed performance gain depends on a particular carefully chosen formulation rather than on the underlying framework. We therefore evaluate whether our method remains stable when the prompt is

Table 16: Stability of Qwen3-8B under four semantics-preserving prompt reformulations. The prompt versions substantially differ in wording, organization, and structural presentation while preserving the same task semantics and information. Prompt-specific results report task performance and execution cost, while the across-prompt statistics summarize the sensitivity of each metric to prompt formulation.

Prompt version	Task performance		Execution and planning	
	TC (%) ↑	SR (%) ↑	Steps ↓	Avg. replans ↓
A. Prompt-specific results				
Original	40.90	10.00	13,808.72	26.21
v1	40.58	10.33	**13,302.18**	24.78
v2	**41.62**	**10.67**	13,342.00	25.02
v3	41.22	**10.67**	13,481.50	**24.58**
B. Across-prompt stability				
Mean	41.08	10.42	13,483.60	25.15
Sample standard deviation	0.44	0.32	229.98	0.73
Coefficient of variation (%)	1.08	3.08	1.71	2.90
Range (max–min)	1.04	0.67	506.54	1.63

substantially reformulated. Starting from the original prompt, we construct three additional versions by randomly modifying its wording, organization, and structural presentation while preserving the same task semantics, task-relevant information, and intended reasoning objective. All four prompt versions are evaluated on the same 300 episodes under otherwise identical settings.

To characterize prompt sensitivity, we report both prompt-specific outcomes and their across-prompt dispersion. For each metric, the sample standard deviation measures the typical variation across prompt realizations, the coefficient of variation normalizes this variation by the corresponding mean, and the range measures the full spread between the best and worst observed formulations.

Table 16 shows that substantial prompt reformulation causes only limited variation in both task performance and execution behavior. None of the alternative formulations produces a systematic degradation relative to the original prompt. Instead, the results remain tightly clustered, with some reformulations slightly improving task performance and others reducing execution or replanning cost. Dispersion remains small for all four metrics, including not only TC and SR but also simulation steps and replanning frequency. Thus, the stability is not limited to final task outcomes. This behavior is consistent with the design of our framework, in which historical organization, evidence retrieval and binding, and multi-agent coordination are represented explicitly by the system. Changing the surface realization of the prompt therefore affects how the backbone receives the reasoning instructions, but does not redefine the underlying historical representation or collaboration mechanism. The observed stability suggests that the resulting behavior is driven primarily by these explicit mechanisms and the preserved task semantics.

C.4.2 Stability across Randomly Sampled Evaluation Suites

The main experiments are conducted on a difficulty-balanced 300-episode evaluation suite sampled from the complete benchmark. A natural concern is whether the reported performance depends on the particular episodes selected for this suite. To examine this sensitivity, we independently construct five additional 300-episode evaluation suites using different random seeds while preserving the same difficulty-balanced sampling protocol. We evaluate both DeepSeek-V4 and Qwen3-8B on every sampled suite and compare them with the main evaluation suite. The purpose of this experiment is not to require identical results across samples, but to determine whether changing the sampled episodes materially alters the performance level or the qualitative conclusions of the evaluation.

We examine task performance together with failure, planning, and execution behavior. TC measures overall task completion and SR measures strict episode-level success. Zero TC denotes episodes in which no task requirement is completed, whereas Nonzero failure denotes episodes that make positive progress but do not achieve strict success; together, they characterize the composition of

Table 17: Stability across independently sampled difficulty-balanced 300-episode evaluation suites. The main suite is the subset used in the primary experiments, while Samples 1–5 are independently constructed using different random seeds under the same sampling protocol. Results compare task performance, failure composition, planning burden, and execution cost for two representative LLM backbones. For Replans / episode, replanning events are summed across all agents within each episode and then averaged over episodes.

Model	Metric	Main suite	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
DeepSeek-V4	TC (%)	55.99	53.12	53.03	53.03	53.30	53.81
	SR (%)	19.00	17.00	18.67	20.00	19.33	18.67
	Zero TC (%)	7.33	8.67	9.67	9.00	11.33	10.33
	Nonzero failure (%)	73.67	74.33	71.67	71.00	69.33	71.00
	Replans / episode	36.89	36.21	35.46	40.28	36.35	38.41
	Simulation steps / episode	10,876.4	13,004.4	13,148.9	13,688.4	12,919.0	13,575.9
Qwen3-8B	TC (%)	40.90	42.95	43.89	43.83	43.56	42.95
	SR (%)	10.00	8.33	11.00	12.00	10.67	11.33
	Zero TC (%)	11.00	8.67	10.33	10.33	13.67	12.67
	Nonzero failure (%)	79.00	83.00	78.67	77.67	75.67	76.00
	Replans / episode	52.41	51.42	49.90	53.49	49.18	51.16
	Simulation steps / episode	13,808.7	13,161.8	12,968.6	13,194.8	12,369.3	13,348.4

unsuccessful execution. Replans and simulator steps provide complementary views of planning burden and embodied-execution cost.

As shown in Table 17, task performance is broadly comparable across the five new evaluation suites. In particular, TC varies within a narrow range across these samples for both backbones, despite each suite containing a different set of episodes. This suggests that the difficulty-balanced sampling procedure maintains a broadly comparable level of challenge across these new samples. The stability is also visible in the composition of failures. Across samples, the proportions of complete stalls and partial-but-unsuccessful executions remain broadly similar. Thus, comparable TC and SR are accompanied by broadly similar mixtures of these failure categories. The sampled suites expose the agents to similar types of execution outcomes, suggesting that the sampling procedure broadly retains the observed pattern of task failure. Planning behavior also shows some consistency. Replanning frequency changes moderately between samples but remains within a consistent operating range for each backbone. Simulator-step cost is somewhat more sensitive to the particular sampled episodes than task completion, which is expected because trajectory length depends on the concrete sequence of navigation, interaction, and recovery required by individual tasks. Overall, these results suggest that some qualitative findings extend beyond a particular 300-episode draw.

C.5 Representative Failure Analysis and Remaining Limitations

Our experiments demonstrate that Tapestry Weaving (TW) and Tapestry Evidence Grounding (TEG) substantially improve historical memory organization, retrieval, and reference binding, supporting more reliable collaborative execution. Nevertheless, semantically complex instructions that combine diverse task requirements remain challenging, even when relevant historical evidence is available. We examine a representative case in which both object groups are retained in **Entailing Evidence**, but the LLM selects the wrong referent amid long context containing numerous weak supporting records. The case illustrates how limitations in downstream reasoning and sensitivity to evidence presentation can persist despite improved memory support. Despite its strong empirical performance, TEG may still fail to account for the full range of semantic information in complex instructions.

In an earlier episode, two structurally similar configurations are created. `book_0` and `laptop_0` end together on `table_1`, while `lamp_0` and `teapot_0` end together on `table_3`. The subsequent instruction asks the agents to

> "Take the two objects that ended together on the long living-room table; switch off the powered one, empty the filled vessel, and move both to the small table the kettle originally came from."

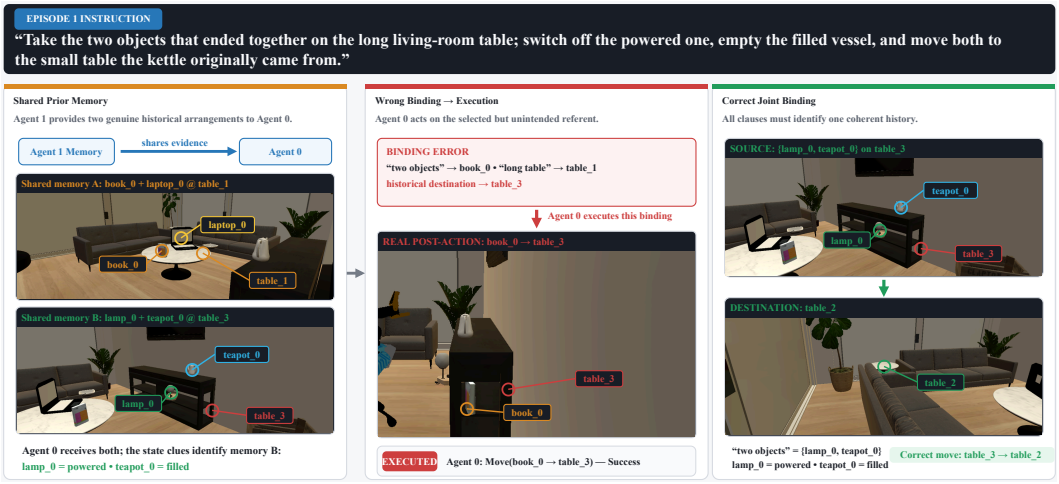

Figure 10: Representative failure under competing historical support and weak-evidence interference. Left: Agent 1 provides Agent 0 with two factually valid prior configurations: `book_0` and `laptop_0` on `table_1`, and `lamp_0` and `teapot_0` on `table_3`; both groups are retained in Entailing Evidence. Middle: the actual erroneous binding selects `book_0` and `table_1`, grounds the historical destination to `table_3`, and leads Agent 0 to successfully execute `book_0`→`table_3`. Right: the intended binding jointly resolves the entity, state, and historical-location constraints, identifying {`lamp_0`, `teapot_0`} on `table_3` as the target entities and `table_2` as the correct destination.

The intended binding identifies {`lamp_0`, `teapot_0`} on `table_3` as the referenced pair and `table_2` as the historical destination. The instruction combines a multi-entity reference with state information and a history-dependent target, while also containing an infeasible sub-operation, namely emptying the filled vessel.

Both configurations are supported by multiple historical records relevant to the instruction's descriptions, including objects occurring together and their table-related histories. This case does not involve same-category instance distractors: the competing groups contain different object types, and neither is routed to **Contradiction Witnesses** as a conflicting same-type alternative. Both groups are retained as candidate alternatives in **Entailing Evidence** and remain available to the downstream LLM rather than being collapsed into a single highest-scoring hypothesis. Membership in this structure indicates sufficient evidence coverage for consideration, not a unique binding or satisfaction of every task requirement.

The failure occurs during the LLM's subsequent interpretation of this evidence-rich context. Alongside the relevant records, the prompt contains numerous weakly informative historical associations. Such evidence need not be false to interfere with selection: it can make competing object and location associations salient without helping distinguish the intended referent. Once the wrong multi-entity configuration is favored, it can also bias the interpretation of the historical destination. The retained history contains several real table-related relations that provide plausible support for competing locations. A selected location may therefore be supported by genuine evidence belonging to a different historical realization. Entity and destination errors can reinforce one another as the LLM interprets these associations together. In this case, the LLM favors the book-related history, selects `book_0` and `table_1`, and resolves the historical destination to `table_3`. Agent 0 then successfully moves `book_0` to `table_3`. The executor's success confirms the physical action, not the correctness of the LLM's reference selection.

Figure 10 contrasts the retained historical configurations, the erroneous binding and executed action, and the intended interpretation. The relevant object group is present in the supplied evidence, but the downstream LLM still chooses the wrong object and location.

This case illustrates a remaining challenge in translating improved memory support into reliable decisions under complex task composition. TW preserves the relevant histories, and TEG provides

structured evidence for grounding, but the final interpretation still depends on the LLM's ability to integrate multiple references and task requirements. The combination of compositional reasoning demands, competing historical associations, and weak evidence in the prompt can limit this process. As instructions combine more diverse operations and historical dependencies, reliable execution therefore requires both effective memory mechanisms and robust downstream reasoning. In a memory-continuous setting, this distinction also matters for subsequent experience reuse.

Addressing these remaining challenges is a central focus of our future work. Building on the improvements in historical organization, retrieval, and binding, we aim to strengthen compositional reasoning for complex instructions, reduce weak-evidence interference through more selective context presentation, and improve interpretation verification and recovery during collaborative execution. Evaluating these capabilities across more diverse tasks will help characterize how model capability, memory use, and execution interact, and extend the benefits of continuous cross-episode collaboration to a broader range of demanding instructions and task settings. In parallel, we view substantially longer task streams and larger accumulated histories as a complementary direction for future evaluation. Contemporary embodied policies still exhibit substantial failures in long-horizon navigation, manipulation, constraint satisfaction, and collaborative execution; as stream length increases, these execution failures and the corresponding corrective interventions can accumulate, making it increasingly difficult to isolate the effects of memory scale itself. We therefore focus the current benchmark on controlled cross-episode continuity, while leaving evaluation over tens or hundreds of episodes to a later stage in which embodied execution is more reliable. As execution capabilities improve, Tapestry can naturally be extended along stream length, historical depth, the number of persistent entities and configurations, and total memory size, enabling cleaner investigation of long-term memory scalability.

D Tapestry Benchmark Construction, Curation, and Difficulty Levels

The **Tapestry Benchmark** evaluates continuous embodied multi-agent collaboration across inter-dependent episodes in evolving 3D environments. Current instructions may depend on latent task states established through prior observations, actions, state transitions, and inter-agent communication. Each sample therefore belongs to a continuous task stream rather than an independent episode, allowing later tasks to reuse previously established entities, locations, functional states, and multi-object configurations, including superseded historical versions. The Tapestry Benchmark construction pipeline builds upon PARTNR; however, given the substantial differences between the two tasks, we introduce several modifications to the original pipeline. The construction process combines model-assisted generation, structured processing, episode-complete human annotation, and Habitat verification. Model assistance expands the compositional coverage of generated tasks; human annotators determine the final task semantics and constraints; and fixed-schema programs instantiate these specifications as resettable, automatically measurable embodied environments.

In real-world embodied task execution, robots typically operate under task-level supervision: the robot first attempts the requested task, while its outcome may subsequently be inspected and any unfinished or incorrectly executed requirements completed or corrected before the next request. Meanwhile, users or other activities may further change the environment between tasks. We follow the same process when constructing continuous task streams. Each episode is first executed by an agent policy, and the resulting environment changes are retained. Any unmet task requirements or invalid execution outcomes are then manually completed or corrected, while preserving valid changes produced by the agent policy. We subsequently apply scene-valid changes to a subset of entities to reflect environmental evolution between tasks. After physical validation, the resulting environment becomes the initial state of the subsequent episode.

Task-level completion or correction and scene-valid inter-task changes modify the environment state, but they are not directly written into agent memory. The retained memory contains only information actually available to the agents through their observations, executed actions, execution outcomes, and inter-agent communication. Therefore, an externally introduced state change may affect the initial environment of a later episode without appearing as a remembered event. Its effect becomes available to an agent only if it is subsequently observed or communicated. This design avoids privileged

historical information and reflects embodied settings in which the world can change outside an agent's own interaction history.

This construction preserves policy-induced environment evolution while preventing unresolved execution failures from disrupting the intended cross-episode dependencies of the task stream. The retained agent history records its observations, actions, execution outcomes, and communication, whereas successive episode states additionally reflect task-level correction and inter-task environmental changes. Together, these components produce continuous task streams that capture the evolving interaction between embodied execution, task supervision, and environmental change.

D.1 Formalizing Tapestry Benchmark Task Streams

Continuous task stream. A benchmark stream is defined as

$$\mathfrak{T} = \left(\sigma, \{\mathcal{E}_e\}_{e=0}^{K-1}, \{s_{e,0}\}_{e=0}^{K-1}\right), \qquad K = 10. \tag{4}$$

Here, σ denotes a fixed multi-room scene, $\mathcal{E}_e$ is the e-th embodied episode, and $s_{e,0}$ is its initial environment state. All ten episodes share the same scene and form a continuous versioned history in which entities can be introduced, transformed, revisited, and reused after multiple intervening changes. The stream remains indivisible during difficulty assignment and data splitting, ensuring that a later task is never separated from the historical context that determines its meaning.

Each episode is instantiated as a cooperative Dec-POMDP,

$$\mathcal{E}_e = \left(\mathcal{S}, \{A_i\}_{i=1}^{N}, T, R, \{\Omega_i\}_{i=1}^{N}, O, \gamma; s_{e,0}, I_e, \mathcal{G}_e\right). \tag{5}$$

$\mathcal{S}$ is the global state space; A_i and Ω_i are the action and observation spaces of agent i; T is the environment transition function; R is the reward; O is the observation function; and γ is the discount factor. The episode-specific terms are the initial state $s_{e,0}$, the history-conditioned instruction I_e, and the structured evaluator $\mathcal{G}_e$. This formulation separates the shared embodied interaction model from the particular environment, instruction, and success program instantiated at each position in the stream.

Policy-driven stream progression. Stream progression follows the task-level supervision process used in real-world embodied task execution. For episode e, a construction-time agent policy π^A first executes instruction I_e from $s_{e,0}$, producing the policy rollout state

$$\tilde{s}_{e,+} = \mathcal{C}\left(s_{e,0}, I_e; \pi^A\right), \tag{6}$$

where $\tilde{s}_{e,+}$ preserves the environment changes actually produced by the agent policy during execution.

The rollout outcome is then inspected against the task requirements. Any unmet requirements or invalid execution outcomes are manually completed or corrected, while valid policy-induced changes are preserved. We denote this correction process by

$$s_{e,+}^{V} = \mathcal{R}\left(\tilde{s}_{e,+}, \mathcal{G}_e\right), \tag{7}$$

where $\mathcal{R}$ denotes the task-level completion and correction operator and $s_{e,+}^{V}$ is the verified post-task environment. Importantly, $\mathcal{R}$ is corrective rather than substitutive: it retains valid changes produced by the policy rollout and intervenes only on unmet task requirements or invalid outcomes required for valid stream progression.

Between successive tasks, a subset of eligible entities is sampled for scene-valid changes in location and/or functional state. Let Δ_e denote these inter-task changes. The next episode begins from

$$s_{e+1,0} = \mathcal{F}\left(s_{e,+}^{V}, \Delta_e\right), \tag{8}$$

where $\mathcal{F}$ applies the sampled changes subject to scene affordances, physical validity, and simulator instantiation constraints. Invalid changes are resampled rather than retained. Thus, environment evolution across episodes combines policy-induced task execution, targeted completion or correction of unresolved outcomes, and additional changes occurring between tasks.

Recurring entities preserve their identities together with ordered location, functional-state, and configuration histories. Earlier versions are retained, allowing later episodes to refer to recent transitions, earlier entity states, or designated historical configurations. The resulting stream therefore preserves both the current environment and the preceding versions from which history-dependent tasks are constructed.

History-conditioned latent task state. For agent i, its retained task and interaction history within episode e is

$$H^i_{e,t} = \left(I_e, o^i_{e,0:t}, a^i_{e,0:t-1}, m^i_{e,0:t-1}\right). \tag{9}$$

Here, I_e is the retained task instruction, and o, a, and m denote local observations, actions, and exchanged messages, respectively; action outcomes and controller feedback are included in o. The accumulated task and interaction history available before episode e is

$$M_{<e} = \bigcup_{k<e} \bigcup_{i=1}^{N} H^i_{k,T_k}. \tag{10}$$

An underspecified instruction is grounded as

$$g_e = \phi\left(I_e, M_{<e}, s_{e,0}\right). \tag{11}$$

The pair $(s_{e,0}, M_{<e})$ therefore constitutes a latent task state: the current environment determines which operations are executable, while retained history determines which entity, state version, source, target, or configuration the current instruction denotes. The two components play complementary roles: $M_{<e}$ records the agents' actual observations, actions, execution outcomes, and communication, whereas $s_{e,0}$ reflects the environment after preceding policy execution, task-level completion or correction, and inter-task evolution. Historical evidence must therefore be interpreted against the current environment.

D.2 Construction of Continuous Task Streams

Each task stream is constructed through a coordinated sequence of scene-grounded task generation, task-level completion or correction, inter-task environment evolution, historical enrichment, physical instantiation, evaluator construction, and human verification. The construction process maintains a persistent record of recurring entities, their state transitions, and previously established configurations across the entire stream. These records are reused when constructing later episodes, allowing the resulting tasks to depend on information distributed over different points in the interaction history while remaining grounded in a concrete Habitat scene.

The construction is deliberately split into semantic, physical, and evaluation stages. Scene metadata first determines which entities, locations, and state changes can be used. Model-assisted generation then proposes a structured sequence of household tasks under this scene constraint. The resulting symbolic tasks are normalized and instantiated into concrete Habitat episodes, after which their intended outcomes are compiled into structured evaluators and finalized through human review. The construction procedure is detailed below through its task-construction, physical-grounding, and evaluation–admission stages.

Scene-grounded task space. Figure 11 summarizes the task-construction stage, including scene-constrained task generation, environment evolution, historical induction, and versioned configuration construction. Construction begins from a scene-specific inventory extracted from the HSSD metadata and simulator interface. For every selected scene, we enumerate its semantic rooms, numbered furniture and receptacles, movable-object categories, available asset families, navigable floor regions, and functional-state affordances. Semantic furniture identifiers are linked to the concrete receptacle handles used by Habitat, while object categories are associated with the asset families from which concrete instances can later be sampled. Cleaning, filling, and power-state operations are enabled only for object classes that support the corresponding state transition.

This inventory is reused throughout the complete construction pipeline. During task generation, it constrains which objects, rooms, furniture items, and state changes can appear in an instruction. During parsing, it provides the vocabulary against which generated entities and aliases are normalized. During physical realization, it determines candidate assets, receptacles, and semantic placement regions. During evaluator construction, it provides the mapping from symbolic task arguments to entities that can be resolved in the simulator.

Navigability is also treated as a scene-level property. The navigation mesh is computed using the configured embodied-agent geometry and the static scene structure, and the valid indoor navigable region is retained for subsequent floor placement. This prevents a symbolic floor target from being

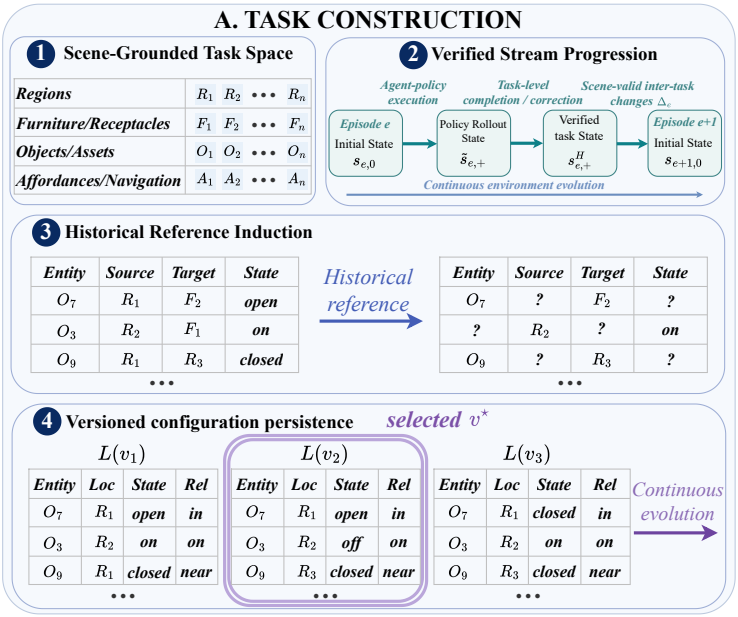

Figure 11: Construction of history-dependent tasks within a continuous task stream. Scene-supported entities and affordances define the grounded task space. Each episode is first executed by an agent policy, producing a policy-induced environment state. Unmet requirements or invalid outcomes are then completed or corrected while preserving valid policy-induced changes, after which controlled inter-task changes produce the environment for the next episode. Recurring entities accumulate ordered location and state histories, from which later historical references and configuration-recovery tasks are constructed.

accepted solely because its room label exists; a physical placement must later lie inside a region that can actually be reached and instantiated.

The first episode is proposed from this grounded task space. The goal is to formulate a household task objective characterized by a complex relational structure, encompassing multiple objects, locations, and supported state transformations.

Continuous stream expansion. After the seed episode, the remaining tasks are generated sequentially while construction maintains the current and historical states of recurring entities. For each recurring entity x, we summarize its task-relevant state at the beginning of episode e as

$$\mathbf{x}_e = (\ell_e(x), q_e(x)), \tag{12}$$

where $\ell_e(x)$ denotes the current location of x and $q_e(x)$ denotes its current functional state. The latter provides a unified representation for the applicable object state maintained by the stream, without requiring a separate variable for every state type.

The construction-time agent policy first executes episode e, producing the policy rollout state of entity x. Valid policy-induced changes are retained, while any unmet task requirements or invalid execution outcomes are subsequently completed or corrected to obtain the verified post-task state:

$$\begin{aligned} \widetilde{\mathbf{x}}_{e,+} &= \left(\widetilde{\ell}_{e,+}(x), \widetilde{q}_{e,+}(x)\right), \\ \mathbf{x}^V_{e,+} &= \left(\ell^V_{e,+}(x), q^V_{e,+}(x)\right). \end{aligned} \tag{13}$$

Here, $\widetilde{\mathbf{x}}_{e,+}$ records the state actually produced by the agent-policy rollout, whereas $\mathbf{x}^V_{e,+}$ denotes the verified state after targeted completion or correction. Manual intervention is corrective rather than substitutive: changes already produced validly by the agent policy are preserved, and only unresolved task requirements or invalid outcomes are modified. Newly introduced entities are added with their resulting verified location and functional-state records.

After task-level completion or correction, a subset of entities and furniture is sampled from the environment and subjected to scene-valid inter-task changes. Let $\widehat{\ell}_e^{\Delta}(x)$ and $\widehat{q}_e^{\Delta}(x)$ denote the sampled location and functional-state updates when the corresponding attribute is changed. The state used to construct the next episode is

$$\ell_{e+1}(x) = \begin{cases} \widehat{\ell}_e^{\Delta}(x), & \text{if } x \text{ receives an inter-task location change,} \\ \ell_{e,+}^{V}(x), & \text{otherwise,} \end{cases}$$
$$q_{e+1}(x) = \begin{cases} \widehat{q}_e^{\Delta}(x), & \text{if } x \text{ receives an inter-task state change,} \\ q_{e,+}^{V}(x), & \text{otherwise.} \end{cases} \tag{14}$$

Only changes compatible with scene affordances and physical instantiation are retained. Consequently, construction of the next episode uses the environment resulting jointly from policy-induced execution, targeted completion or correction, and intervening environmental changes.

To preserve how each entity evolves across the constructed stream, every episode transition is appended to an ordered construction-side entity history,

$$\mathcal{P}_{\leq e}(x) = \mathcal{P}_{<e}(x) \ \oplus \ \left[e,\, \mathbf{x}_e \to \widetilde{\mathbf{x}}_{e,+} \to \mathbf{x}_{e,+}^{V} \to \mathbf{x}_{e+1}\right], \tag{15}$$

where $\oplus$ denotes temporal append. The appended record distinguishes the state produced by agent execution, the subsequent task-level completion or correction, and the inter-task environment change. This record is construction metadata used to preserve environment provenance and is not added to the agents' interaction memory: $M_{<e}$ contains only the agents' actual observations, actions, execution outcomes, and communication. Hence, $\mathbf{x}_{e+1}$ provides the latest environment state used when constructing later episodes, while $\mathcal{P}_{\leq e}(x)$ preserves the ordered sequence of earlier environment versions and their transition provenance. Later construction can therefore use the current entity state while tracing a requested location or functional state back to the point in the stream at which that historical version was established. Figure 11 visualizes the policy-induced state transition, task-level correction, inter-task evolution, and ordered history used during stream expansion.

Subsequent task construction uses these entity-level records together with the stored configuration history and recent task context. Each new episode preferentially reuses established entities and may introduce additional scene-valid objects when useful. Across the ten-episode horizon, recurring entities accumulate multiple location and state versions, while earlier versions remain available for delayed historical references. Multi-entity configurations are maintained in parallel and provide the corresponding version history for relational structures.

Historical Dependency Construction. Historical dependence is introduced after the underlying task semantics have been grounded to information already established in the stream. Candidate references are derived from recorded entity identities, previous locations, state transitions, source and target relations, and temporal provenance. Selected information can then be omitted from the surface form of a later instruction while its antecedent remains explicitly recorded in the construction metadata. As illustrated in Figure 11, this abstraction can hide selected entity, source, target, or state information while retaining its historical binding in the construction record.

For every induced historical reference, the construction retains the concrete historical binding that resolves it. This record includes the referred entity or entity set, the relevant state version, the historical episode or transition providing the evidence, and the semantic role that the recovered information plays in the current operation. References whose intended antecedent cannot be recovered unambiguously from the retained stream records are not admitted as valid history-dependent constructions.

The effective dependency horizon follows the state lineage associated with the requested information. The most recent occurrence of an entity can differ from the historical point that determines the requested source, location, or state. Consequently, later episodes can depend on information several transitions earlier even when the entity itself has appeared more recently. This construction creates history dependence through recorded state evolution rather than through lexical distance alone.

Within-Episode Continuity and Cross-Episode Continuity are retained as separate sources of dependence during audit and evaluation. The former covers both within-instruction references and dependencies on observations, actions, outcomes, or object states established earlier in the current episode, whereas the latter requires retrieving information established before the current episode.

Versioned configuration persistence. A multi-entity configuration records the states and relations of a group of entities at a particular point in the stream. Once such a configuration is established, later policy execution, task-level completion or correction, or inter-task evolution may move or change any subset of its members. These changes produce newer configuration versions, while earlier versions are retained in the construction history. The same group of entities can therefore accumulate a sequence of layouts with different locations, functional states, and spatial relations as the stream progresses.

For a later configuration-recovery task, construction first selects one previously stored version as the recovery target and then generates the instruction and evaluator from that version:

$$v^\star = \mathrm{Select}(\mathcal{L}_{<e}), \qquad (I_e, \mathcal{G}_e) = \mathrm{Construct}\left(\mathcal{L}^{(v^\star)}, \mathbf{s}_{e,0}\right), \tag{16}$$

where $\mathcal{L}_{<e}$ denotes the configuration versions established before episode e, $\mathcal{L}^{(v^\star)}$ is the selected historical version, and $s_{e,0}$ provides the current stream context in which the recovery task is instantiated.

The selected version fixes the entities, states, and relations that the later task is intended to recover. The instruction can subsequently omit part of this configuration information or refer to it through its stored layout identity, while the evaluator preserves the complete target derived from $\mathcal{L}^{(v^\star)}$. Intermediate episodes may have produced newer states for some of the same entities, but those changes do not overwrite the earlier version selected for recovery. Configuration Continuity tasks therefore require reconstructing a coherent historical arrangement whose members and relations were jointly valid at the same point in the stream. Figure 11 makes this construction order explicit: multiple configuration versions are retained during continuous evolution, and one historical version $v^\star$ is selected before the corresponding recovery task is constructed.

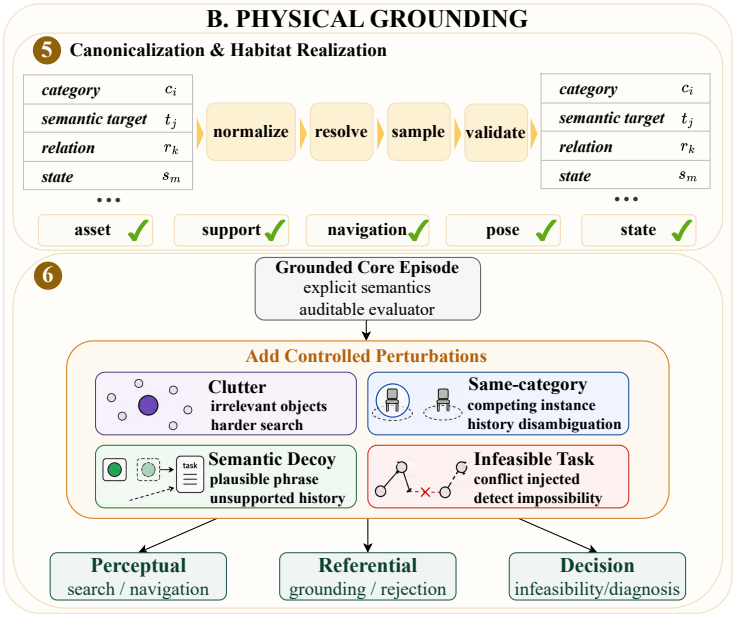

Figure 12: Physical grounding and controlled stress construction. Structured task specifications are canonicalized, resolved to simulator entities, sampled, and validated to form a grounded core episode. Controlled clutter, same-category competitors, semantic decoys, and infeasible-task conflicts are then introduced to probe perceptual, referential, and decision-level robustness.

Canonicalization and Habitat realization. The physical-grounding stage is summarized in Figure 12. Before simulation, every generated task is normalized into a common structured schema. Object and furniture aliases are mapped to canonical names, required fields and multiplicities are checked, and all rooms, object classes, receptacles, action types, functional-state assignments, and

ordering expressions are validated against the scene inventory. Unsupported actions, malformed state descriptions, invalid object–state combinations, and exact duplicate instructions are rejected or conservatively repaired. Historical expressions are retained only when their antecedents have already been recorded in the stream.

The normalized symbolic task is then instantiated into a concrete Habitat episode. Each semantic object category is mapped to an available object asset, while each furniture target is resolved to compatible receptacle handles in the selected scene. Objects assigned to receptacles are sampled on valid support surfaces. Floor-assigned objects are sampled inside the designated semantic region and restricted to navigable locations. Initial functional states are subsequently applied to the instantiated objects.

Physical sampling checks several conditions that cannot be established from scene metadata alone. A selected asset must be available in the scene configuration; its target support must admit the intended placement; the pose must remain stable and collision-consistent; floor placements must be navigable; and the requested functional state must be applicable to the chosen asset. Bounded resampling is used when a valid symbolic task encounters an incidental asset, support, or pose failure. An episode is rejected when no valid realization can be obtained within the permitted attempts.

The sampler retains detailed construction provenance, including selected asset templates, semantic selectors, simulator handles, receptacles or floor regions, sampled poses, applied functional states, number of attempts, and failure causes. These records separate physical instantiation problems from semantic task errors and from execution failures observed later during policy evaluation.

During benchmark construction, each semantic entity is assigned a stream-unique construction-side entity key, represented internally by an object name with a numeric index. These keys are retained in construction metadata, historical-reference annotations, evaluator compilation, and stream-level consistency records. Instructions express targets through object names and descriptions, and agents infer the target binding from instruction semantics, observations, and retained history. Sampling preferentially reuses representative asset families and placement characteristics for recurring entities, maintaining controlled appearance continuity while allowing distinct instances for multiplicity or interference.

Controlled clutter, interference, and infeasibility stress. After the core episode has been grounded, the controlled perturbations in Figure 12 are introduced to probe three complementary forms of robustness: perceptual search, referential grounding, and decision-level infeasibility recognition.

Common-sense clutter augments the instantiated scene with task-irrelevant objects selected from scene-compatible categories. Task-relevant categories are protected from accidental substitution, and every additional object passes the same asset and placement validation used during ordinary Habitat instantiation. Clutter therefore increases visual search, navigation, and scene density while leaving the task evaluator unchanged.

Same-category interference introduces a more targeted source of ambiguity. Using a random seed, approximately 30% of eligible task-relevant initial-state entries are selected, with at least one candidate whenever a valid entry exists. A competing instance from the same semantic category is then instantiated at a different valid furniture–region context available in the episode. The injected instance must occupy a distinguishable scene-valid location and pass the standard asset, support, pose, and reset checks. Correct grounding therefore depends on historically anchored information such as prior location, state changes, or movement provenance. Naturally occurring duplicates and clutter may create additional alternatives, and the final interference audit retains only instances that remain physically distinguishable in the realized scene.

Semantic-decoy interference starts from a valid task; construction adds extra, linguistically plausible phrases whose suggested historical interpretations lack support in the recorded history. These phrases supplement rather than replace the legitimate task descriptions and historical references. They are drawn from nearby object semantics and plausible room or furniture descriptions while preserving instruction fluency. The original task requirements and gold historical referents remain unchanged; the added phrases are distractors, not additional task requirements. Correct behavior is to reject their unsupported interpretations without generating concrete bindings, while retaining valid bindings and continuing the underlying task.

A separate transformation constructs infeasible tasks. Here the goal is to preserve the outward form of an executable embodied instruction while deliberately violating one or more conditions required for successful completion. The injected conflict may arise from unavailable skills, impossible spatial relations, invalid execution order, or incompatible layout requirements. Conflicts are injected at this stage; diagnostic labels are confirmed after evaluator compilation, specifying whether infeasibility exists and, when applicable, its failure type and associated target. This produces tasks that test whether a system can distinguish executable historical instructions from ones that are internally inconsistent or physically unrealizable, and whether it can localize the source of failure rather than merely fail during execution.

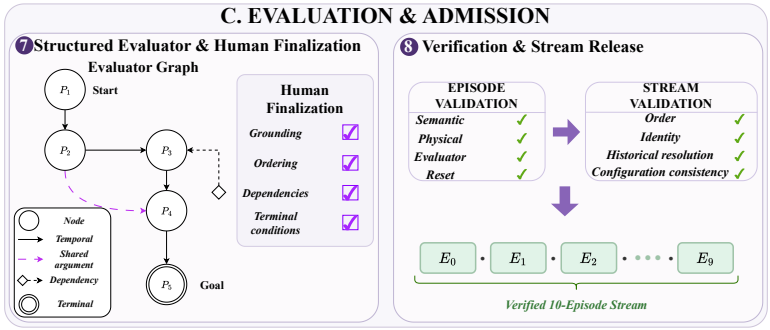

Figure 13: Evaluation, human finalization, and stream-level admission. The structured evaluator encodes task propositions and their temporal, dependency, shared-argument, and terminal structure, which is finalized through human review. Episode-level validity is verified before stream-level order, identity, historical resolution, and configuration consistency are checked; only complete verified ten-episode streams are admitted to the benchmark.

Structured evaluator and human finalization. Figure 13 summarizes the final evaluation–admission stage. Each physically instantiated episode is compiled into a structured evaluator whose proposition program refers directly to the entities sampled in Habitat. Placement requirements are represented through support, containment, floor, and room predicates; functional requirements encode all kinds of states; and relative-spatial requirements connect multiple concrete entities.

The evaluator additionally stores execution structure that cannot be recovered from terminal propositions alone. Temporal edges specify valid precedence relations among proposition groups. Activation dependencies introduce requirements after their prerequisites have been satisfied, while maintenance dependencies require previously established conditions to remain valid during later operations. Argument-sharing constraints force designated propositions to use the same object or target instance, and terminal constraints determine which conditions must hold at task completion.

Compilation first produces a grounded draft from the structured instruction and the instantiated Habitat entities. Every released episode is subsequently reviewed together with its instruction, initial world state, preceding stream history, historical-reference annotations, and evaluator draft. Human finalization verifies semantic-to-entity grounding and determines the final temporal order, dependencies, shared arguments, and terminal requirements.

All finalized semantic arguments are then resolved to the simulator handles of the released episode. This produces the executable evaluator used for online truth-value updates, task completion measurement, success determination, and the diagnostic annotations associated with historical grounding and constraint auditing.

Verification and stream release. After evaluator finalization, admission follows the two-level verification procedure shown in Figure 13: episode validity is checked first, followed by consistency of the complete continuous stream.

At the episode level, semantic validation checks that the instruction, historical references, and finalized evaluator describe the same intended operation. Physical validation checks that all task-relevant objects, supports, regions, poses, and functional states have been successfully instantiated. Evaluator

validation requires every proposition and constraint argument to resolve to a live simulator entity and all dependency references to point to valid evaluator elements. Reset validation confirms that the episode can be restored to its recorded initial state and that the initial proposition truth values can be computed successfully.

Failures are handled according to their source. Deterministic naming or schema issues can be repaired when the intended semantics remain unchanged. Physical sampling failures can trigger bounded resampling. Invalid evaluator references are recompiled after safe grounding corrections or cause the episode to be rejected. Controlled infeasibility cases used by the reliability audit are retained under explicit negative annotations and remain separate from ordinary generation or reset failures.

Stream-level verification then checks the complete stream package as a whole. All ten episode indices must be present exactly once and remain in temporal episode order. Recurring entity identities must agree with the stored episode-boundary environment records, and a historical reference must resolve to an entity, state, event, or configuration already established in an earlier part of the same stream. Versioned configuration records are checked for consistent membership and provenance, and the complete stream must remain within one scene and one difficulty partition.

Only verified complete streams enter calibration and experimental splitting. The admitted unit is therefore the complete verified ten-episode stream shown in Figure 13, preserving its episode order and historical dependencies as a single benchmark sample. The retained construction package preserves the scene inventory, episode-boundary environment states and inter-task transition records, persistent entity histories, configuration versions, historical-reference annotations, physical sampling traces, finalized evaluators, curation records, and verification metadata. These records provide the hidden structure required for automatic scoring and benchmark audit.

The policy-facing task remains substantially narrower. Evaluated agents receive the natural-language instruction, egocentric observations, action outcomes, communication interface, and the interaction history permitted by the experimental protocol. Construction does not assign evaluator propositions to individual agents and does not prescribe a task decomposition or communication schedule. Allocation, coordination, information exchange, and memory management therefore remain decisions of the evaluated multi-agent system.

D.3 Agent Interface and Executable Capability Envelope

Feasibility is defined relative to the skills exposed by the evaluation configuration, not every operation that the simulator could implement internally. The planner selects high-level skills over targets resolved from its current observations and task context; skill controllers then use geometry, proprioception, and simulator state to execute path following, grasping, placement, articulation, or functional-state changes. Controller access to navigation and collision data does not expose global scene state to the planner. The policy receives the configured observations, scene description, and action outcomes, but never the evaluator propositions, gold bindings, or conflict annotations. At execution time, observed object instances are entered into the policy-visible scene representation with runtime-assigned instance identifiers. These identifiers are created by the multi-agent system solely to distinguish represented physical instances and to provide unambiguous references for planning, communication, and skill execution. Table 18 lists the interface, principal preconditions, and returned information.

The complete team executes one shared packed episode, while the evaluator specifies task conditions without assigning propositions to particular agents. One agent may observe or transform an entity in one room, and another may need that record in a later episode. Communication can therefore support immediate subtask allocation and dependency handoff as well as preservation of cross-episode evidence. The benchmark specifies which observations and messages are available under the protocol, but does not encode a hidden label for which agent must discover or complete each requirement.

D.4 Construction of Continuity and Reliability Requirements

The three continuity dependencies describe how task-relevant information must be preserved and reused over progressively longer temporal horizons. **Within-Episode Continuity** supports both the interpretation of later subtasks and the tracking of execution dependencies along a continuous task trajectory within a single episode. Later subtasks may depend on observations, actions, execution

Table 18: High-level agent capabilities and their low-level execution substrate. Task feasibility is defined against this interface; privileged geometry used inside a controller is not policy access to evaluator-only state.

Layer	Planner-facing interface	Preconditions and effect	Feedback or visibility
Observation and scene representation	RGB, depth, panoptic, and proprioceptive observations; agent-specific scene description	Adds observed rooms, furniture, movable instances, support or room relations, holding state, and applicable functional states to the local representation	Exposes policy-observable scene information together with runtime-assigned instance identifiers; privileged simulator and evaluator metadata remain unavailable to the planner.
Navigation and exploration	`Navigate[target]`, `Explore[room]`, `Wait[]`	Approaches an observed target, visits its candidate furniture in a room to expand coverage, or remains idle for coordination	Returns completion or an explicit target, path, or reachability failure.
Object transport	`Pick[object]`, `Place[object, relation, target, ...]`, `Rearrange[...]`	Pick requires a reachable movable object and a free gripper; Place requires a held object, a supported relation, and a valid pose; Rearrange composes navigation, pick, and place	Returns grasp or release success, a precondition failure, or a placement-sampling failure.
Articulation	`Open[furniture]`, `Close[furniture]`	The target must be an observed articulated entity within interaction range and free of state conditions that block the operation	Returns joint-action completion or a validation, state, or reachability failure.
Functional state	`Clean[object]`, `Fill[object]`, `PowerOn[object]`, `PowerOff[object]`	Metadata determines whether the state is supported; cleaning and filling additionally enforce distance and applicable source conditions, such as access to water	Returns success or a capability, distance, source, or state-precondition failure.
Low-level control	Navigation paths, base velocities, grasp/release commands, articulated joints, and sampled poses	Performs reachability, support or containment, snap-down, and collision checks before committing the physical action	Returns controller diagnostics to the high-level planner without exposing privileged geometry directly.
Team coordination	Protocol-permitted task, action, progress, observation, held-object, and failure updates	Supports coordination within an episode and reuse of permitted teammate records in later episodes	Adds teammate updates to the receiving context without gold answers or evaluator labels.

outcomes, or object states established earlier in the same episode. At the same time, a later operation may become available only after an earlier proposition is satisfied, or an established support or room condition may need to remain true while a subsequent change is made. Same-argument constraints prevent two same-category instances from separately satisfying clauses intended for one entity. Executing all atomic actions in an arbitrary order, destroying a required condition prematurely, or switching to another same-category instance therefore cannot receive full credit.

Cross-Episode Continuity arises from the continuous evolution of entities, locations, and functional states across episodes, leaving at least one operation argument recoverable from accumulated cross-episode experience. The antecedent need not be an explicit state transition: it may directly provide an

Table 19: Release-side continuity audit counts. The 238 Configuration Continuity episodes include both establishment and invocation cues.

Operational audit view	Release
Within-Episode Continuity evidence	1,388 (86.8%)
Cross-Episode Continuity evidence	1,385 (86.6%)
Configuration Continuity participation	238 (14.9%)

entity identity, a previous support or room, a functional state, an operation source or target, or the time at which a relation held. Requesting the correct historical variable, retrieving a related record, and grounding the complete variable to the correct entity set are distinct outcomes. A record that mentions the correct category may still be insufficient if it omits the location, state version, or one of the required members.

Configuration Continuity extends cross-episode dependencies over longer horizons through evolving scene configurations and requires the agent to recover the designated historical configuration $\mathcal{L}^{(v^\star)}$ selected during task construction. As intervening episodes modify individual members, the stream can accumulate multiple configuration versions, each recording member identities, locations, functional states, and spatial relations. Solving the current task therefore requires aligning the relevant entity histories to the common version $v^\star$ rather than resolving each entity independently.

The target version jointly determines the member identities, their historical states, and the relations that must hold among them. Information observed at different times can be individually valid while remaining mutually incompatible as one configuration. Recovery is successful only when the retrieved entity histories collectively reconstruct the version encoded by $\mathcal{L}^{(v^\star)}$, preserving both its member states and cross-entity relations.

Release-side continuity audits are conservative operational flags rather than exhaustive definitions of the three dependencies. As a conservative surface proxy, the Within-Episode Continuity audit checks whether an eligible anaphoric expression has a compatible earlier entity mention within a multi-character local window. After episode 0, the Cross-Episode Continuity audit checks explicit historical, repetition, or alias cues, or a referential expression without a compatible antecedent in the current instruction but with a compatible antecedent in the preceding stream history; layout-only evidence is excluded. The Configuration Continuity audit checks an early named multi-entity layout or a late name-only invocation backed by at least two stored members without reenumerated member operations. Table 19 reports the resulting views.

Reliability requirements follow the decision path from historical interpretation to embodied execution. Let $\widehat{\mathcal{B}}_e$ denote the selected entity binding and $\widehat{g}_e$ the resulting grounded operation. The complete path is

$$(I_e, M_{<e}) \longrightarrow \widehat{\mathcal{B}}_e \longrightarrow \widehat{g}_e \longrightarrow \mathrm{Feasible}(\widehat{g}_e, s_{e,0}) \longrightarrow \mathrm{execution}. \tag{17}$$

This path separates four questions: whether the system recognized the need for a historical variable, whether it selected the intended physical instance, whether the candidate interpretation was supported by recorded history, and whether the grounded operation was feasible under the active capabilities and geometry. Table 20 summarizes the controlled conditions and diagnostics.

Instance Disambiguation is resolved by historical location, state, and movement evidence rather than category identity alone. When an expression legitimately denotes several entities, the gold annotation retains the full set, so Exact Selection does not impose a single-target assumption. Evidence Validation targets the added unsupported phrases, not the legitimate historical dependencies of the underlying task. False Need measures unnecessary historical requests induced by these phrases; Variable Leakage records their promotion into spurious historical variables in the operational context; and False Binding records concrete entity bindings assigned to the unsupported interpretations. Retrieval and binding required by valid task dependencies are not errors merely because the episode contains a decoy. The desired behavior is rejection of the added interpretations while continuing the valid task; `UNRESOLVED` is one possible runtime expression of this rejection.

Table 20: Historical grounding and the three reliability conditions. Gold negative conflict records are evaluated separately from ordinary positive episode success.

Evaluation view	Controlled condition and gold behavior	Scope	Diagnostics
Overall historical grounding	Identify the historical variables that must be requested and resolve them to the complete accepted entity assignment(s)	Full release: 8,805 minimal historical requirements and 10,320 entity assignments	Retrieval-Request Recall and E2E Exact Grounding.
Instance Disambiguation	Same-category candidates occupy distinct scene-valid locations; retrieve the complete gold set and select no distractor	Fixed 30% transformation, plus eligible natural duplicates and clutter	Gold-Set Retrieval, Exact Selection, and Distractor Capture.
Evidence Validation	Added phrases lack historical support; reject their bindings while preserving valid task grounding and execution	100 manually adjudicated semantic-decoy episodes; valid underlying tasks remain executable	False Need, Variable Leakage, and False Binding.
Constraint-Conflict Auditing	A grounded task conflicts with skills, terminal requirements, spatial anchors, or sampled geometry; recognize infeasibility and localize its cause	599 atomic conflicts over 291 episodes from 86 streams	Precision, Recall, F1, FPR, and Exact Hit; downstream TC, Zero-Prog., and Budget-Exh.

Constraint-conflict audit annotations are finalized after evaluator compilation for tasks with previously injected conflicts. The release audit contains 345 capability–state conflicts, 208 pinned-anchor cross-room conflicts, 22 terminal-location conflicts, 21 simulator-confirmed adjacency conflicts, and 3 required-target placements for which no valid pose can be sampled, totaling 599 atomic conflicts. Initial-state instantiation failure remains a generation failure; the last audit family instead describes an episode that can reset but whose required target placement is infeasible. Exact Hit requires the correct conflict type and, for an object-specific gold record, the associated object. TC, Zero-Prog., and Budget-Exh. after an attempted execution describe the consequences of false acceptance rather than replacing diagnostic quality.

D.5 Evaluator Compilation, Automatic Metrics, and Verification

Natural-language instructions are ultimately compiled into evaluator programs that reference concrete simulator handles. Model-proposed propositions and candidate temporal structure are first converted to the fixed predicate and constraint vocabulary. Annotators finalize the program against the instruction, stream history, and sampled initial state. The packer then revalidates the schema, resolves semantic objects and furniture to episode handles, and imports delayed-layout requirements. Runtime scoring uses this human-finalized, handle-grounded program rather than model-generated prose. We write the evaluator as

$$\mathcal{G}_e = (\mathcal{P}_e, \mathcal{D}_e, \Psi_e, \mathcal{C}_e)\,. \tag{18}$$

$\mathcal{P}_e$ is the grounded proposition set, $\mathcal{D}_e$ the temporal DAG, Ψ_e the activation and maintenance dependencies, and $\mathcal{C}_e$ the same-argument, different-argument, and terminal constraints. The predicate inventory is reported in Table 21. Placement predicates distinguish support, containment, floor, and room relations while retaining concrete instance handles; functional-state predicates apply only to categories whose metadata supports the corresponding state.

The release contains 45,466 temporal-DAG edges and 7,744 explicit proposition dependencies: 4,248 `after_satisfied` and 3,496 `while_satisfied`. A later placement may depend on the most recent prior placement of the same entity, and a state change may depend on the most recent contradictory state of the same object. A next-to relation that shares an entity with a placement in the same temporal group may require that placement to remain satisfied; floor placement and its room

Table 21: Evaluator proposition inventory in the 1,600-episode release.

Predicate family	Count	Included predicates
Support and room	13,389	on-top, inside, on-floor, in-room
Relative spatial	6,822	next-to
Cleanliness	1,822	clean, dirty
Filledness	1,282	filled, empty
Power	1,269	powered-on, powered-off
Total	24,584	

condition receive the analogous maintenance semantics. Terminal constraints remove intermediate source states that have been superseded by later changes, so those states must occur in the correct order but need not remain true at episode end. An additional 130 same-argument constraints require selected propositions to share an object or target, preventing different same-category instances from satisfying clauses intended for one entity.

At each simulator step, the evaluator updates proposition truth values and dependency availability. Let $\mathrm{Sat}_e(p)$ indicate that proposition p has been satisfied under its applicable constraints. Episode Task Completion is

$$\mathrm{TC}_e = \frac{1}{|\mathcal{P}_e|} \sum_{p \in \mathcal{P}_e} \mathbf{1}[\mathrm{Sat}_e(p)] \,. \tag{19}$$

Strict Success Rate is

$$\mathrm{SR}_e = \mathbf{1}[\forall p \in \mathcal{P}_e, \mathrm{Sat}_e(p)] \,. \tag{20}$$

TC measures the fraction of atomic requirements completed under the applicable constraints; SR is one only when the complete episode objective is satisfied. Task-Completion Steps and Replanning Frequency are computed from the embodied trace. Constraint-coverage analysis additionally groups episodes into spatial (S), temporal (T), constraint-free (F), and mixed (M) regimes according to their active constraints. This coverage grouping is separate from predicate families and memory audit flags.

Semantic correctness, physical resettability, and evaluator integrity cannot be established by one check. Each episode therefore passes the layered gates in Table 22. Human review establishes the intended interpretation, fixed-schema checks validate structure and references, and simulator reset confirms that the initial state can be instantiated. Only streams whose ten episodes all pass episode-level checks proceed to calibration and splitting.

D.6 Structural Difficulty Stratification and Dataset Composition

We distinguish episode-level scoring, stream-level difficulty assignment, and level-wise summary statistics. We first score each of the ten episodes in a task stream using its task structure and historical context. We then manually assign the complete stream to a difficulty level, keeping its episodes together because later tasks depend on earlier states, configurations, and events. Only after this assignment do we compute the mean episode score within each level, reported as "Struct." in Table 24.

Structural difficulty score. For episode e in stream $\mathfrak{T}$, the ten-point structural score combines memory complexity, spatial density, temporal structure, and their joint coupling:

$$S^{(e)}_{\mathrm{struct}}(\mathfrak{T}) = S_{\mathrm{mem}}(\mathfrak{T}) + S^{(e)}_{\mathrm{spa}}(\mathfrak{T}) + S^{(e)}_{\mathrm{temp}}(\mathfrak{T}) + S^{(e)}_{\mathrm{coup}}(\mathfrak{T}), \qquad S^{(e)}_{\mathrm{struct}}(\mathfrak{T}) \in [0, 10]. \tag{21}$$

Spatial and temporal scores are computed from each episode's own evaluator counts, and the coupling term uses those episode-level scores. The memory term retains the stream-level construction signal

Table 22: Human, programmatic, and simulator verification gates. Failures to instantiate an initial state are separated from controlled negative conflicts retained in compiled tasks.

Gate	Verified condition	Failure handling or retained record
Schema and scene vocabulary	Required fields parse; structured objects and furniture resolve to the scene vocabulary; alias and supported-action filters are applied	Apply only implemented deterministic canonicalization, otherwise reject.
Human semantic and constraint review	Instruction, stream context, initial state, historical antecedents, propositions, temporal structure, dependencies, shared arguments, and terminal requirements jointly express the intended task	Correct or rewrite before packing; every released episode passes complete review.
Initial-state sampling and reset	Assets, supports, floor or receptacle poses, functional states, simulator reset, and initial metrics can be produced	Record attempts and failure reasons, then regenerate upstream or exclude.
Evaluator handle integrity	Object arguments, receptacle arguments, and both sides of spatial relations resolve to live simulator entities; dependency and constraint indices exist	Reground through the supported packing path, otherwise reject.
Argument-constraint integrity	Fields named by same- or different-argument constraints are actually produced by the referenced proposition results	Record the failed proposition and argument and reverify after correction.
Stream completeness	Episode indices are unique and ordered, and exactly ten packed episodes are present	Do not admit incomplete streams to calibration or splitting.
Independent feasibility audit	Capability direction, terminal location, pinned spatial anchors, adjacency geometry, and required-target placement are checked after compilation	Retain controlled conflicts in the negative-audit view rather than mixing them with ordinary positive tasks.
Post-task and inter-task transition validation	Agent-policy execution is recorded; any unmet requirements or invalid outcomes are corrected to produce a valid post-task state; sampled inter-task changes are scene-valid and yield an instantiable next-episode environment.	Complete or correct only failed requirements while preserving valid policy-induced changes; resample invalid inter-task changes or reject the transition if no valid realization can be obtained.

specified in Table 23, so episode scoring does not discard the historical context of the stream. The threshold rules are applied before computing the level-wise averages; they are not applied to counts first averaged across the ten episodes.

Level assignment. After scoring all ten episodes, we manually assign each complete stream $\mathfrak{T}$ to a structural difficulty level $L(\mathfrak{T}) \in \{1, \ldots, 5\}$ using these scores as references for reviewing its task structure and historical demands. This is a score-informed manual assignment, not automatic binning by a fixed aggregate score or cutoffs. Difficulty assignment operates on complete streams rather than assigning their episodes independently, preserving every task's historical antecedents. The resulting stream-level labels are frozen for dataset accounting and evaluation-suite construction.

Difficulty labels are therefore determined by the structure of the task stream itself. Same-category interference and the separately annotated reliability stresses are likewise excluded from the difficulty score; they are evaluated as orthogonal robustness factors.

Table 23: Deterministic components of the ten-point episode-level structural difficulty score. Spatial, temporal, and coupling scores are computed for each episode; the memory term retains its stream-level construction context and does not replace the multi-label memory annotations. All terms are independent of evaluated-policy performance.

Component	Deterministic scoring rule	Points
Memory complexity	`short_term`: 0; `long_layout_restore`: 2 when the construction-side `long_layout_episode_count` ≥ 5	0 or 2
Spatial density	For the episode's next-to count N: $N = 0$: 0; $0 < N < 3$: 1; $3 \leq N < 8$: 2; $N \geq 8$: 3	0–3
Temporal structure	Sequential rule over the episode's temporal-edge count E and explicit-dependency count R: $E < 10$ and $R < 2$: 0; else $E < 35$ and $R < 8$: 1; else $E < 70$ and $R < 15$: 2; otherwise: 3	0–3
Spatial–temporal coupling	2 when the spatial score is 3 and the temporal score is at least 2; otherwise 1 when both scores are nonzero; otherwise 0	0–2

Table 24: Structural characterization of the five complexity levels after stream-level grouping. "Struct." is the mean ten-point episode score over all episodes assigned to each level, as defined in Eq. 22. Propositions, next-to relations, temporal edges, and dependencies are also averaged per episode within each level.

Level	Streams	Episodes	Struct.	Props./ep.	Next-to/ep.	Temp. edges/ep.	Deps./ep.
1	34	340	2.44	9.29	1.14	10.62	1.34
2	34	340	3.21	11.16	2.31	15.15	2.25
3	34	340	4.12	14.40	3.44	29.71	3.66
4	34	340	5.68	16.85	5.83	30.98	6.64
5	24	240	8.50	29.19	10.41	66.95	12.60

Structural characterization of the levels. Let $\mathcal{D}_\ell = \{\mathfrak{T} : L(\mathfrak{T}) = \ell\}$ be the set of streams assigned to level ℓ. After grouping, we report

$$\text{Struct.}_\ell = \frac{1}{10|\mathcal{D}_\ell|} \sum_{\mathfrak{T} \in \mathcal{D}_\ell} \sum_{e=0}^{9} S^{(e)}_{\text{struct}}(\mathfrak{T}). \tag{22}$$

This is the mean score over all episodes in the level.

Table 24 also reports mean per-episode next-to relations, temporal edges, explicit dependencies, and evaluator propositions within each level. These statistics describe the resulting groups. Proposition count provides a complementary view of overall task-program size.

Dataset composition and balanced evaluation. The full release contains 160 complete streams, corresponding to 1,600 episodes across 15 scenes. Under the frozen, manually assigned difficulty partition, Levels 1–5 contain 34, 34, 34, 34, and 24 streams, respectively.

The evaluation protocol selects 30 complete streams, with six streams from each difficulty level, yielding a balanced suite of 300 episodes. Selection is performed at the stream level so that every evaluated episode retains the preceding history from the same continuous stream. The resulting evaluation set therefore balances structural difficulty without breaking the historical dependencies that define later tasks.

Dataset composition and balanced evaluation. The full release contains 160 complete streams, corresponding to 1,600 episodes across 15 scenes. Under the frozen, manually assigned difficulty partition, Levels 1–5 contain 34, 34, 34, 34, and 24 streams, respectively.

Table 25: Stream-level accounting for the full release and the difficulty-balanced evaluation suite.

	Full release		Evaluation suite	
Level	Streams	Episodes	Streams	Episodes
1	34	340	6	60
2	34	340	6	60
3	34	340	6	60
4	34	340	6	60
5	24	240	6	60
Total	160	1,600	30	300

Table 26: Stream-level accounting for the full release and the difficulty-balanced evaluation suite.

	Full release		Evaluation suite	
Level	Streams	Episodes	Streams	Episodes
1	34	340	6	60
2	34	340	6	60
3	34	340	6	60
4	34	340	6	60
5	24	240	6	60
Total	160	1,600	30	300

The evaluation protocol selects 30 complete streams, with six streams from each difficulty level, yielding a balanced suite of 300 episodes. Selection is performed at the stream level so that every evaluated episode retains the preceding history from the same continuous stream. The resulting evaluation set therefore balances structural difficulty without breaking the historical dependencies that define later tasks.

Historical-grounding evaluation additionally contains 1,610 minimal historical binding requirements with 1,779 entity assignments. A requirement corresponds to an independently resolvable historical expression, while one coordinated expression can refer to multiple entities, producing more assignments than requirements. Pure within-instruction anaphora, explicit simulator identifiers, generic task themes, and redundant subphrases are excluded from this count. These annotations are evaluator-side records used for historical request, retrieval, and entity-assignment diagnostics and are never exposed to the evaluated policy.

E Prompt Templates

This appendix summarizes the prompt interfaces of Tapestry Reasoning, connecting instruction interpretation, historical-memory use, multi-agent coordination, and embodied action selection. Generic backend messages and baseline planner prompts are omitted because they do not define Tapestry Reasoning. We focus on the information provided to each reasoning component, its expected structured output, and how that output is passed to subsequent stages.

The system follows decentralized execution. All agents share the same prompt structure but receive agent-specific context, including identity, role, active-stage assignment, local world state, execution feedback, and validated teammate updates. Each agent therefore plans from its own policy-visible context rather than assuming shared observations. Current ownership assignments and teammate progress are included to avoid duplicated or conflicting operations. This shared-template design can be instantiated for different configured team sizes without defining agent-specific prompt variants.

Historical reasoning is handled by Tapestry Weaving (TW) and Tapestry Evidence Grounding (TEG). TW organizes available historical records into queryable tapestry evidence. TEG identifies unresolved historical dependencies required by the active instruction and converts them into structured retrieval conditions. References resolvable from the current instruction or current state are handled without historical retrieval. Retrieved evidence may complete only an identified dependency and cannot

introduce new task requirements. TEG ranks candidate hypotheses by evidence coverage and organizes them into Entailing Evidence, Contradiction Witnesses, and Contextual Evidence. The LLM compares the retained candidate groups using the instruction, current context, and structured evidence; an interpretation it judges supported and executable is passed to action planning, while an unsupported dependency remains unresolved.

The resulting context is passed to decentralized action selection. At each planning step, the acting agent receives the active stage contract, local world graph, relevant historical evidence, teammate updates, and executor feedback, and proposes one grounded action. A stage-clarification interface constructs the compact stage contract while preserving explicitly specified entities, targets, relations, and requested states. Text enclosed by «...» denotes runtime-inserted values. The prompt interfaces below are organized by functional role: decentralized action selection, historical-dependency routing, TEG instruction completion, and stage clarification.

E.1 Decentralized action selection

P1: Decentralized action-selection prompt

```
    1. Current stage exact task
    <<active_stage_id_and_instruction_or_completion_notice>>

    2. Negotiated multi-agent assignment
    <<current_object_or_stage_ownership; otherwise non-
duplication guidance>>

    3. Observed objects, furniture, rooms, and states
    <<observed_target_state>>
    <<grounded_object_location_and_room_candidates>>
    Current-agent grounding: use only runtime-assigned instance
identifiers already associated with entities represented in the
current observation or local world graph. If the required object
or furniture has not yet been assigned such an identifier, gather
additional observations instead of inventing one.
    Local observed world graph:
    <<partial_world_graph_for_this_agent>>

    4. Other-agent broadcast
    <<compact_teammate_intent_progress_and_observation_updates>>

    5. Previous execution feedback
    Previous action: <<last_skill_call_or_none>>
    Execution success: <<yes_no_or_not_finished>>
    Failure reason: <<executor_feedback_or_none>>

    6. General planning principles
    Choose one executable action for this agent. Use grounded
entities from the
    current context, coordinate with the teammate's reported work,
 and use
    execution feedback when revising the plan. Do not add goals
that are absent
    from the instruction or its resolved historical context.

    You are Agent_<<agent_id>> in a decentralized multi-agent
household task.
    Choose one executable low-level skill call for yourself.
    <<task_giver_or_receiver_role>>

    Available skills and argument formats:
    - Navigate[target_id]
    - Explore[room_id]
    - Wait[]
```

```
    - Pick[object_id]
    - Place[object_id, relation, target_location_id,
spatial_constraint,
    reference_id]
    - Rearrange[object_id, relation, target_location_id,
spatial_constraint,
    reference_id]
    - Clean[object_id], Fill[object_id], PowerOn[object_id],
PowerOff[object_id].
    - Open[entity_id], Close[entity_id].
    - EndStage[stage_id]
    - Done[]

    Select an action supported by the current instruction and
environment context.
    Use STAGE_COMPLETE=yes only when available evidence indicates
 that the current
    stage is complete.

    Reply exactly in this format:
    STAGE_COMPLETE: yes|no
    Agent_<<agent_id>>_Action: <Skill>[<arguments>]
    IMPOSSIBLE_SUBTASKS: none|<detected impossible subtasks>
    IMPOSSIBLE_ACTIONS: none|<detected impossible actions>
    Assigned!
```

P1a: Structured teammate update injected into P1

```
    Agent_<<receiver_id>>_TeammateUpdate:;
    from agent_<<sender_id>>;
    <<intent_or_progress>>;
    <<optional_completed_skill_and_status>>;
    <<optional_next_skill_intent>>;
    <<optional_held_object>>;
    <<up_to_three_observed_entity_deltas>>;
    <<optional_ready_stage_and_stage_statuses>>
```

E.2 HISTORICAL-DEPENDENCY ROUTING

P2: Whole-instruction historical-dependency detection

```
    Determine whether completing the instruction requires
historical information
    that is unavailable in the instruction, resolved context, or
current
    environment state. Return JSON only.

    Schema:
    {"needs_memory_retrieval":true|false,
    "reason":"short reason",
    "historical_dependencies":["instruction phrases requiring
history"],
    "current_context_dependencies":["phrases resolvable from
current context"]}

    Task:
    - Identify information whose interpretation depends on prior
interactions.
```

```
    - Do not request memory when the current instruction and
environment context
    already provide enough information for execution.
    - Historical wording alone is not a dependency unless it
leaves task-relevant
    information unresolved.
    - Identify the unresolved phrase without predicting its
answer.

    Full instruction: <<full_instruction>>
    Resolved instruction context: <<resolved_instruction_context
>>
    Current environment state: <<environment_state_summary>>
```

P3: Current-instruction reference resolution

```
    Resolve references using only the current instruction and its
 preceding stages.
    Return JSON only.

    Schema:
    {"resolved_stage":"stage with current-instruction references
resolved",
    "local_bindings":[{"phrase":"copied phrase",
                                                "binding":"entity or
target established by current context",
                                                "source":"previous
stage used"}],
    "unresolved_memory_phrases":["phrases still requiring
historical memory"],
    "needs_historical_memory_after_local_resolution":true|false,
    "reason":"short reason"}

    Rules:
    - Use only the full current instruction, the previous stages
listed below, and
    the current stage.
    - Resolve references when their meaning is established by the
 current
    instruction context.
    - Keep references that genuinely require historical evidence
unchanged and
    list them in
    unresolved_memory_phrases.
    - Preserve explicit task requirements and do not introduce
new ones.

    Previous stages in the current instruction:
    <<previous_stage_text_and_optional_refinements>>
    Current stage S<<stage_id>>: <<current_stage>>
```

P4: Stage-level historical-dependency detection

```
    Determine whether the current stage contains unresolved
historical
    dependencies. Return JSON only.

    Schema:
```

```
    {"needs_memory_retrieval":true|false,
    "reason":"short reason",
    "historical_dependencies":["unresolved phrases requiring
history"],
    "currently_resolvable_phrases":["phrases resolved by current
context"]}

    Task:
    - Use the resolved instruction context and current
environment state before
    requesting historical evidence.
    - Request memory only for information needed to complete the
stage that remains
    unresolved in the current context.
    - Preserve the stage semantics and identify dependencies
without grounding
    them at this step.

    Original full task: <<full_instruction>>
    Resolved instruction context: <<resolved_instruction_context
>>
    Current stage: <<current_stage>>
    Current environment state: <<environment_state_summary>>
```

E.3 TEG INSTRUCTION COMPLETION

The interfaces below follow candidate construction and evidence organization in Eq. 3. Before LLM-based completion, TEG places clearly conflicting same-type alternatives in Contradiction Witnesses, retains the two highest-coverage candidate groups whose scores are strictly greater than one half in Entailing Evidence, and assigns the remaining lower-coverage or lower-ranked candidates to Contextual Evidence. These structures remain distinct: coverage-based ranking supplies candidate alternatives for the LLM's subsequent consistency and executability assessment.

P5: TEG structured retrieval-condition construction

```
    TEG completes history-conditioned instructions. Given the
current instruction
    and environment context, identify the historical dependencies
 required for
    task completion. Convert each dependency into a structured
retrieval condition
    based on the instruction semantics, resolved context, and
current state.

    Return JSON only:
    {"retrieval_conditions":[{
    "condition_id":"C<id>",
    "dependency_phrase":"phrase copied from the instruction",
    "required_information":"entity|entity_set|location|state|
action_history|relation|configuration",
    "temporal_scope":"relevant historical context",
    "semantic_constraints":{"entity_type":[],"location_type":[],
                                                    "
required_state":[],"required_relation":[]},
    "resolved_context":"relevant information already established",

    "reason":"why historical evidence is required"
    }]}

    Guidelines:
```

```
    - Create a retrieval condition only when historical evidence
is necessary for
    completing the instruction.
    - Preserve the dependency phrase and explicit semantic
constraints.
    - Describe the information that must be retrieved without
predicting its value.
    - Use resolved context and current state to avoid redundant
conditions.
    - Represent jointly referenced entities together when they
form one semantic
    dependency; otherwise keep independently resolvable
dependencies separate.
    - Return an empty list when no historical dependency remains.

    Current instruction: <<current_stage_instruction>>
    Resolved context: <<resolved_instruction_context>>
    Current state: <<environment_state_summary>>
```

P6: TEG evidence-based condition completion

```
    Complete one structured retrieval condition using the current
 instruction,
    resolved context, current state, and retrieved historical
evidence. Use
    historical evidence only for the dependency represented by
this condition.

    Return JSON only:
    {"condition_id":"C<id>",
    "completed_instruction_fragment":"resolved fragment or
UNRESOLVED",
    "bindings":[{"dependency":"historical dependency",
                              "resolved_value":"evidence-supported
value or UNRESOLVED",
                              "evidence_ids":["evidence_id"],
                              "confidence":"high|medium|low"}],
    "reason":"short evidence-based explanation"}

    Rules:
    - Preserve explicit requirements from the current instruction.

    - Compare the candidate groups in Entailing Evidence; their
coverage ranking
    is not a final binding decision.
    - Use Contradiction Witnesses to distinguish conflicting same-
type alternatives.
    Contextual Evidence is auxiliary and is not sufficient on its
 own for binding.
    - Select a value only when it is supported by relevant
historical evidence and
    consistent with the resolved context and current state.
    - Return UNRESOLVED when the available evidence is
insufficient or conflicting.
    - Do not introduce entities, relations, states, or actions
absent from the
    instruction and supporting evidence.

    Structured retrieval condition:
    <<retrieval_condition>>
```

```
        Relevant TW historical evidence:
        <<conditioned_historical_evidence>>

        Current instruction:
        <<current_stage_instruction>>

        Resolved context:
        <<resolved_instruction_context>>

        Current state:
        <<environment_state_summary>>
```

P7: TEG-completed context for action planning

```
        History-conditioned context for Agent <<agent_id>>.
        Use the completed context together with the current
    instruction and environment
        state. Preserve unresolved dependencies instead of guessing.

        Current instruction:
        <<current_stage_instruction>>

        Structured retrieval conditions:
        <<retrieval_conditions_or_none>>

        Relevant TW historical evidence:
        <<conditioned_historical_evidence_or_none>>

        TEG-completed instruction fragments:
        <<completed_instruction_fragments_or_none>>

        Resolved bindings: <<phrase_to_entity_bindings_if_any>>

        Use only evidence-supported completions when planning. The
    current instruction
        and environment state remain authoritative for explicitly
    specified facts.
```

E.4 Stage clarification prompt

P8: Stage instruction clarification

```
        Clarify the current stage into executable operations. Use TEG-
    completed
        context when the stage contains a historical dependency.
    Return JSON only.

        Schema:
        {"is_executable": true|false,
        "operations":[{"entity":"entity phrase or grounded identifier
    ",
                                        "operation":"requested
    operation",
                                        "target":"optional target",
                                        "relation":"optional relation
    ",
```

```
                                        "state":"optional requested
    state"}],
        "non_executable_reason":"reason or empty",
        "unresolved_dependencies":["information still unresolved"]}

        Rules:
        - Preserve the task semantics, explicit entities, requested
    states, targets,
        and relations from the current stage.
        - Represent each required operation once and preserve its
    logical order.
        - Distinguish executable task requirements from descriptive
    context.
        - Use retrieved evidence only to complete identified
    historical dependencies.
        - Keep a dependency unresolved when the available context
    does not support a
        unique interpretation, and do not add new task requirements.

        Original full task: <<full_instruction>>
        Current stage: <<stage_instruction>>
        Resolved context: <<resolved_instruction_context>>
        Current state: <<environment_state_summary>>
        TEG-completed context: <<completed_historical_context_or_none
    >>
```