HiP-LR: Hierarchical Planning with Localized Failure Recovery for Long-Horizon Multi-Robot Tasks

Anonymous ACL submission

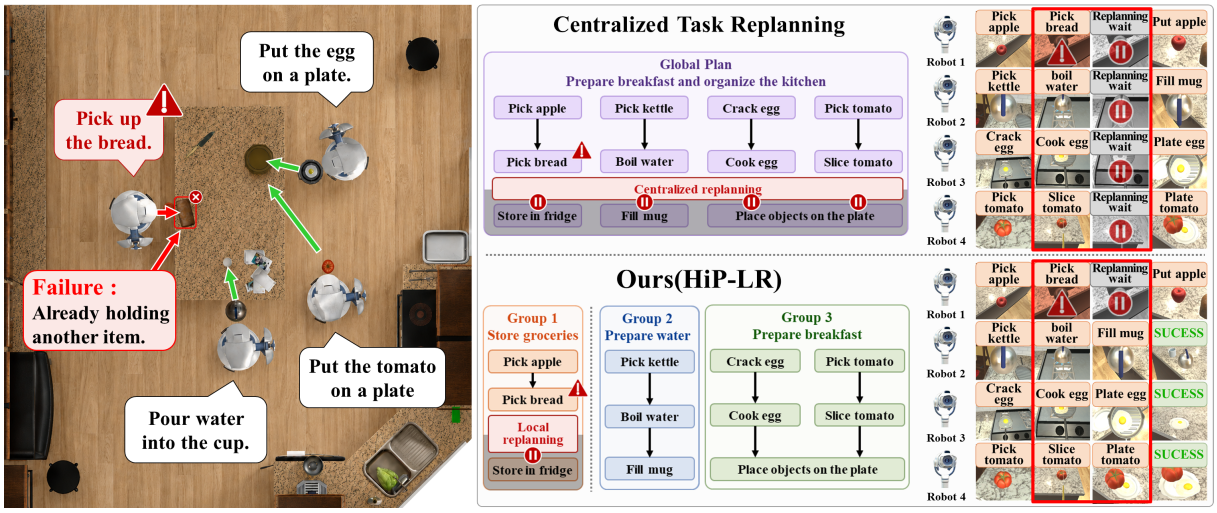

Figure 1: Example of failure-localized replanning in HiP-LR. Upon failure (e.g., occupied gripper), centralized methods (Shek et al., 2026) regenerate the global plan, halting all robots. Conversely, HiP-LR replans exclusively within the affected dependency-isolated group, allowing unaffected robots to continue executing in parallel.

Abstract

Large Language Model (LLM)-driven multi-robot systems struggle with long-horizon tasks due to planning and execution complexities. As task dependencies grow, centralized planners suffer from context explosion and hallucinations, while minor runtime errors can disrupt the entire operation. Current monolithic methods overload LLMs and cause unnecessary global halts during error recovery. To address this, we introduce Hierarchical Planning with Localized Failure Recovery (HiP-LR). HiP-LR uses a two-level hierarchy to divide complex instructions into dependency-isolated subtask groups. A central agent allocates global tasks, while local agents execute specific plans, keeping LLM contexts compact. Furthermore, a PDDL-based dual-memory structure enables failure-localized replanning: if an error occurs, only the affected group pauses to replan, allowing others to continue uninterrupted. Evaluated on MAP-THOR, HiP-LR improves the task success rate by 38% and reduces replan-wait costs by 90% over state-of-the-art baselines. Furthermore, HiP-LR demonstrates highly robust recovery capabilities on our novel failure-injection benchmark, RECAP-THOR.

1 Introduction

Multi-robot systems capable of following natural-language instructions are rapidly transitioning to real-world deployment. The advent of large language models (LLMs) has streamlined this process by decomposing instructions into executable multi-robot plans (Zhang et al., 2024; Kannan et al., 2023; Mandi et al., 2024; Liu et al., 2025). However, long-horizon tasks across multiple robots remain challenging, as difficulty accumulates both when generating the plan and when executing it. During planning, growing subtasks, robot states, and inter-task dependencies expand the planner's context, weakening LLM reasoning and increasing the risk of hallucinated subtasks and infeasible robot-task assignments (Liu et al., 2024; Levy et al., 2024). During execution, even minor failures, such as a missed grasp or an unexpected object displacement, can invalidate downstream subtasks and disrupt the

entire plan. Task success rates therefore degrade in long-horizon settings.

To address these planning and execution challenges, existing studies generally pursue two directions. The first prevents invalid action sequences by constraining LLM outputs with formal logic or explicit structures, such as Planning Domain Description Language (PDDL) (Aeronautiques et al., 1998; Zhang et al., 2025; Shi et al., 2025) or dependency graphs (Obata et al., 2024; Wang et al., 2024). The second direction focuses on execution-time feedback and failure recovery. Frameworks like KGLAMP (Shek et al., 2026), H-AIM (Zeng and Li, 2026), and LLaMAR (Nayak et al., 2024a) monitor runtime errors or violated preconditions and trigger reactive replanning to recover from execution failures.

Yet both directions leave the fundamental challenge of multi-robot planning unresolved. As more robots act on shared objects and spaces, the causal, temporal, and resource dependencies between subtasks accumulate quickly. Prior methods absorb all these dependencies into a single global plan, leaving no internal boundary that separates one source of dependency from another. The complexity is therefore not resolved structurally but pushed entirely onto a single LLM, whose burden grows with every added robot or subtask.

This absence of dependency-aware isolation produces two compounding symptoms. On the planning side, encoding all these dependencies into a single LLM prompt expands the context and weakens reasoning, leading to hallucinated subtasks and infeasible assignments. On the execution side, a minor local failure triggers regeneration of the entire plan and forces every robot to halt, including those unrelated to the failed subtask, as shown in the top panel of Figure 1. The resulting idle time becomes a major source of multi-robot inefficiency.

To address this challenge, we propose **Hierarchical Planning with Localized Failure Recovery (HiP-LR)**, a novel framework that introduces a two-level hierarchy into multi-robot LLM planning. Globally, HiP-LR analyzes these dependencies between subtasks and partitions the task into dependency-isolated groups, as illustrated in the bottom panel of Figure 1. Locally, each group is handled by its own local plan, reasoning only about the subtasks within that group. The LLM's context therefore stays compact and relevant to a single group, distributing the overload that prior methods place on one global LLM across multiple bounded local plans.

A key strength of HiP-LR is *failure-localized replanning*. Because each group is bounded by its own causal, temporal, and resource dependencies, a runtime failure stays inside the affected group. Only that group is suspended and replanned, while independent groups continue executing uninterrupted. To make this localized replanning precise and fast, each group maintains a PDDL-based memory of symbolic states and effects rather than natural-language logs. HiP-LR thereby avoids the replan-wait cost caused by global halts while preserving global consistency.

Our main contributions are outlined as follows:

- We propose **HiP-LR**, a hierarchical LLM-based framework that decouples global task reasoning from local execution via a Central Agent and multiple Local Agents.
- We introduce a dependency-aware grouping strategy that groups causally related subtasks into isolated groups, effectively mitigating context explosion in long-horizon planning.
- We develop a failure-localized replanning mechanism that targets only the affected subtask group, enabling unaffected robots to continue without global execution halts.
- On the MAP-THOR benchmark (Nayak et al., 2024b), HiP-LR reduces replan-wait cost by **90%** and improves the task success rate by **38%** over the SOTA baseline. Furthermore, we introduce and evaluate our framework on RECAP-THOR, a new failure-injection benchmark. Both the benchmark and source code are publicly released.

2 Related Work

Classical planning methods (Georgievski and Aiello, 2015; Browne et al., 2012; Tantakoun et al., 2025; Liu et al., 2025) have long been used for task planning, but they face scalability challenges in open-domain multi-robot settings. Recent LLM-based approaches have improved flexibility by decomposing natural-language instructions, assigning roles, and coordinating robot teams through dialogue or modular agent structures (Zhang et al., 2024; Kannan et al., 2023; Mandi et al., 2024; ichter et al., 2023). However, these methods primarily focus on generating initial plans, leaving

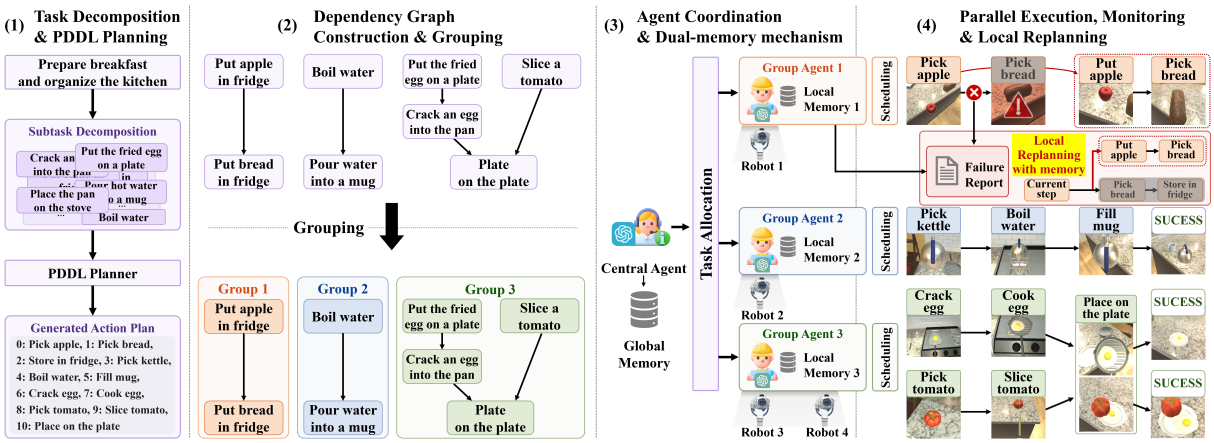

Figure 2: Overview of the HiP-LR framework. (1) HiP-LR decomposes a long-horizon instruction into subtasks solved via PDDL planning. (2) These subtasks are structured into a dependency graph to establish execution dependencies and parallel groups. (3) The Central Agent allocates these groups to Local Agents for distributed scheduling and execution. (4) Local Agents monitor progress via dedicated memory; upon failure, the affected agent generates a PDDL report triggering localized replanning, leaving independent groups uninterrupted.

subtask dependencies and causal ordering implicitly represented. This lack of explicit structure compromises execution-time robustness, particularly in long-horizon tasks where a local failure can invalidate downstream actions. To address these limitations, subsequent studies have explored structured task representations alongside feedback-driven replanning mechanisms for failure recovery.

Structured Reasoning for Long-Horizon Planning. Recent studies improve the reliability of LLM-based planning by grounding plans in explicit symbolic or graph-structured representations for single-agent (Silver et al., 2024; Zhou et al., 2024; Song et al., 2023) and multi-robot settings (Zhang et al., 2025; Obata et al., 2024; Gupta et al., 2025; Bai et al., 2024). For instance, LaMMA-P (Zhang et al., 2025) frames subtasks as PDDL instances solved via Fast Downward (Helmert, 2006) for symbolic validity, while LiP-LLM (Obata et al., 2024) optimizes task ordering and allocation using a dependency graph and linear programming. Constraining LLM outputs with these formal structures reduces hallucinations, invalid constraints, and infeasible assignments. However, because these methods maintain plans as monolithic structures, they struggle to isolate runtime failures and lack a principled mechanism for localized recovery.

Feedback-Loop-Based Replanning. Another line of work introduces execution-time feedback to handle runtime failures (Shek et al., 2026; Borate et al., 2025; Zeng and Li, 2026; Zhu et al., 2025; Son et al., 2025). Reactive methods like EmboTeam (Zeng and Li, 2026) recover through predefined action-level responses, but fail to revise the global plan when errors alter downstream preconditions. Conversely, replanning methods like KGLAMP (Shek et al., 2026) and CoMuRoS (Borate et al., 2025) regenerate the plan upon detecting execution errors. KGLAMP grounds replanning in updated knowledge graphs, while CoMuRoS supports task reassignment under unexpected events.

Despite these advances, lacking intermediate boundaries to isolate failures, current methods often default to global centralized replanning. This inevitably halts all multi-robot execution to prevent state inconsistencies, forcing unaffected robots to idle, an unaddressed inefficiency we define as the replan-wait cost. HiP-LR resolves this limitation through causal isolation. By partitioning tasks into PDDL-derived dependency groups, HiP-LR confines both the failure impact and LLM replanning context strictly to the affected scope. Consequently, this enables localized replanning, allowing unaffected groups to continue uninterrupted and effectively minimizing the replan-wait cost.

3 Method

3.1 Overview

We study the cooperative multi-agent planning problem (Kannan et al., 2023) for long-horizon embodied tasks. Given a natural-language instruction $\mathcal{I}$, an initial state s_0, and a set of robots $\mathcal{R} = \{r_1, \ldots, r_N\}$, the objective is to generate an executable multi-robot plan Π that assigns temporally ordered subtasks and primitive actions to

Algorithm 1 HiP-LR Algorithm

Require: Task instruction $\mathcal{I}$, robot set $\mathcal{R}$, central agent A^{cent}, local agents $\{A_k^{\text{local}}\}$, memories $\mathcal{M}^{\text{global}}$, $\{\mathcal{M}_k^{\text{local}}\}$
Ensure: Executable multi-robot plan and execution status

▷ **Global planning by the Central Agent**
1: $\{T_i\}_{i=1}^{M} \leftarrow \textsc{GetSubtasks}(\mathcal{I}, A^{\text{cent}})$
2: $G^T \leftarrow \textsc{GetSubtaskDG}(\{T_i\}_{i=1}^{M}, A^{\text{cent}})$
3: $\{\mathcal{G}_k^T\}_{k=1}^{K} \leftarrow \textsc{GetSubtaskGroups}(G^T, A^{\text{cent}})$
4: $\{A_k^{\text{local}}\}_{k=1}^{K} \leftarrow \textsc{AssignLoAgent}(\{\mathcal{G}_k^T\}_{k=1}^{K}, A^{\text{cent}})$

▷ **Parallel local planning and execution**
5: **for all** $k \in \{1, \ldots, K\}$ **in parallel do**
6: **repeat**
7: $\mathcal{G}_k^T \leftarrow \textsc{RefineDG}(\mathcal{G}_k^T, A_k^{\text{local}}, \mathcal{M}_k^{\text{local}})$
8: $\mathcal{R}_k \leftarrow \textsc{AssignRobots}(\mathcal{G}_k^T, \mathcal{R}, A^{\text{cent}})$
9: $\mathcal{G}_k^A \leftarrow \textsc{GetActionDG}(\mathcal{G}_k^T, \mathcal{R}_k, A_k^{\text{local}})$
10: **repeat**
11: $s_k \leftarrow \textsc{Execute}(\mathcal{G}_k^A, \mathcal{R}_k, A_k^{\text{local}}, \mathcal{M}_k^{\text{local}})$
12: **until** $s_k = \textsc{Failure}$ **or** $s_k = \textsc{Finished}$
13: $\mathcal{R}, \mathcal{M}^{\text{global}} \leftarrow \textsc{ReportIdle}(\mathcal{R}_k, A^{\text{cent}})$
14: **until** $s_k = \textsc{Finished}$
15: **end for**
16: **return** $\{s_k\}_{k=1}^{K}$

appropriate robots.

As illustrated in Figure 2, we propose HiP-LR, a hierarchical framework that decomposes the overall decision-making process into a global Central Agent and multiple Local Agents. Here, an "agent" refers to an LLM-based reasoning module, not a physical robot. The Central Agent decomposes instructions into subtasks and builds a dependency graph mapping their causal, temporal, and resource relations. It then partitions the graph into independent clusters of mutually dependent subtasks. Finally, each cluster is assigned to a corresponding Local Agent, which refines the local dependency structure, allocates robots, and generates an action-level execution schedule.

3.2 Global Planning with Central Agent

Algorithm 1 (Lines 1–4) outlines the Central Agent's global planning. Given an instruction $\mathcal{I}$ and initial state s_0, an LLM decomposer generates subtasks $\mathcal{T} = \{T_1, \ldots, T_M\}$ detailing required skills, objects, and goals (Line 1). The Central Agent grounds each T_i into a PDDL (Aeronautiques et al., 1998) problem instance $P_i = \langle Objects_i, Init_i, Goal_i \rangle$ using a shared domain model. Solving P_i with the Fast Downward planner (Helmert, 2006) yields a feasible action sequence. Thus, each subtask is represented as a PDDL-grounded model:

$$m_i = \langle \text{pre}(T_i), \text{eff}(T_i), \text{obj}(T_i), \text{res}(T_i), \alpha_i \rangle$$

where $\text{pre}(T_i)$ and $\text{eff}(T_i)$ are symbolic preconditions and effects, $\text{obj}(T_i)$ and $\text{res}(T_i)$ are involved objects and exclusive resources, and α_i is the planned action sequence. Full domain and problem file examples are provided in Appendix A.

Next, the Central Agent constructs a subtask-level dependency graph $G^T = (\mathcal{T}, \mathcal{E})$ (Line 2). Instead of expanding all primitive actions (Obata et al., 2024), HiP-LR infers dependencies among grounded subtasks. The graph construction prompt is invoked once per task, using compact summaries of each subtask, including key objects, preconditions, effects, and goals. When PDDL predicates alone are insufficient to capture semantic ordering, the overall task goal is used as additional context.

Each node $T_i \in \mathcal{T}$ denotes a subtask. Each directed edge $e_{ij} \in \mathcal{E}$ encodes an ordering constraint from T_i to T_j. HiP-LR uses two dependency types:

- *Causal dependency:* An edge e_{ij} is inserted when an effect of T_i satisfies a precondition of T_j, i.e., $\text{eff}(T_i) \cap \text{pre}(T_j) \neq \emptyset$. The edge is directed from the enabling subtask to the dependent subtask.
- *Resource dependency:* An edge e_{ij} is inserted when two subtasks share an exclusive resource, such as the same manipulable object, openable instance, or constrained robot resource. This prevents physical conflicts and inconsistent states during concurrent execution. The edge direction follows the initial PDDL plan order, preserving acyclicity without exhaustive search.

To avoid over-constraining the graph, HiP-LR applies instance-level reasoning. Different instances of the same type do not induce dependencies. Shared open surfaces also do not create edges by themselves. Generic robot predicates and shared source locations are ignored unless they affect execution feasibility. Thus, G^T captures dependencies relevant to execution order and failure containment. Physical interactions not represented in the graph are handled later during execution and replanning. See Appendix B for the full construction procedure.

After constructing G^T, each weakly connected component of G^T defines a subtask group:

$$\mathcal{G}_k^T = (\mathcal{T}_k, \mathcal{E}_k), \quad k = 1, \ldots, K.$$

Because no edge crosses component boundaries, a failure inside $\mathcal{G}_k^T$ cannot structurally propagate to another group $\mathcal{G}_{k'}^T$. Each group therefore serves as a local replanning boundary. Assigning $\mathcal{G}_k^T$ to a Local Agent A_k^{local} (Line 4) establishes an isolated replanning unit.

3.3 Local Scheduling with Local Agent

Once assigned a subtask group $\mathcal{G}_k^T = (\mathcal{T}_k, \mathcal{E}_k)$, the Local Agent A_k^{local} exclusively manages its scheduling and execution. Restricting the LLM's scope to $\mathcal{T}_k$ ensures the prompt contains only relevant context. This mitigates context explosion (Liu et al., 2024; Levy et al., 2024) and enables the generation of a detailed local schedule π_k.

Algorithm 1 (Lines 5–15) outlines parallel local execution. When a group becomes executable, the Local Agent refines its dependency structure to expose parallelism (Line 7). Crucially, PDDL-derived causal and resource edges are strictly preserved as hard constraints, maintaining the global failure boundaries.

The Central Agent mediates robot assignment to prevent conflicts. It allocates a subset of robots $\mathcal{R}_k \subseteq \mathcal{R}$ based on system-wide availability tracked in the global memory (Line 8), cleanly separating global resource management from local reasoning.

Given $\mathcal{R}_k$, the Local Agent expands $\mathcal{G}_k^T$ into an action-level dependency graph $\mathcal{G}_k^A$ (Line 9). Subtasks T_i are replaced by their PDDL action sequences $\alpha_i = \langle a_1^i, \ldots, a_{n_i}^i \rangle$. Cross-subtask edges connect a predecessor's final action to a successor's first, while intra-robot edges serialize a single robot's actions. This yields an executable schedule for immediate action dispatch.

During execution (Lines 10–12), the Local Agent monitors $\mathcal{G}_k^A$ and updates its local memory $\mathcal{M}_k^{\text{local}}$ with successful PDDL effects. This compact local state efficiently tracks subtask statuses and histories. If an execution failure occurs, the Local Agent suspends only its assigned group $\mathcal{G}_k^T$ and utilizes this local memory to trigger a localized replanning loop, ensuring unaffected groups continue uninterrupted.

Failure Detection and Local Replanning. Execution failures arise when outcomes violate PDDL effects or robots report unexecutable actions. Upon detecting a failure in $T_i \in \mathcal{T}_k$, the Local Agent A_k^{local} logs it in $\mathcal{M}_k^{\text{local}}$ and suspends only the affected group $\mathcal{G}_k^T$. Because group-binding dependencies do not cross boundaries, the failure cannot propagate. Meanwhile, the Central Agent continues managing inter-group constraints via global memory $\mathcal{M}^{\text{global}}$.

To initiate recovery, the Local Agent constructs a PDDL-grounded failure report:

$$F_k = \langle T_{\text{fail}}, r_{\text{cause}}, H_{\text{action}}, \mathcal{T}_{\text{remain}}, s_k^{\text{curr}} \rangle,$$

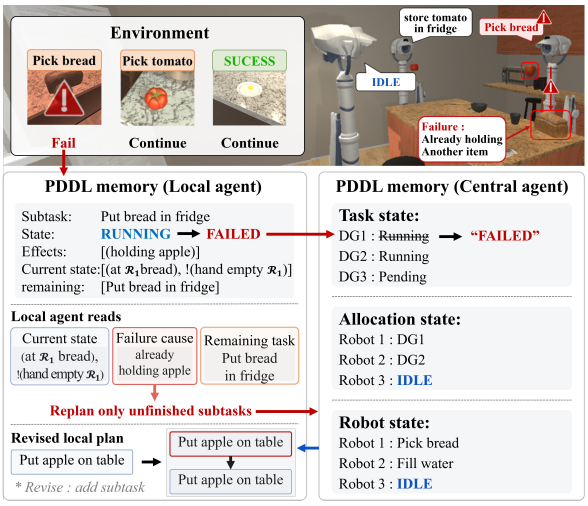

Figure 3: Illustration of HiP-LR's PDDL-based dual-memory update for localized failure recovery. The Local Agent logs the failure state, cause, and pending subtasks in local memory, while the Central Agent updates affected group and robot statuses, leaving independent groups executable.

where T_{fail} is the failed subtask, r_{cause} is the observed failure cause, H_{action} logs successful primitive actions, $\mathcal{T}_{\text{remain}} \subseteq \mathcal{T}_k$ contains unfinished subtasks, and s_k^{curr} is the updated local symbolic state reflecting completed actions. This report provides the minimal symbolic context necessary to replan without losing execution progress.

Crucially, HiP-LR decouples replanning from resource queries. Replanning is confined to $\mathcal{T}_{\text{remain}}$, while queries for newly available robots access $\mathcal{M}^{\text{global}}$. Guided by F_k, the Local Agent re-invokes the PDDL pipeline solely on unfinished subtasks, bypassing completed actions. This yields updated dependency graphs $(\mathcal{G}_k^{T,\text{new}}, \mathcal{G}_k^{A,\text{new}})$ and a revised schedule π_k^{new}. The affected group then resumes execution, leaving independent groups completely uninterrupted.

PDDL-based Dual-Memory Structure. Rather than using natural-language logs (Zhang et al., 2025), HiP-LR's memory uses the symbolic representation employed for planning. Upon an action's success, its PDDL effects are written directly as ground predicates (e.g., `(holding robot1 pan)`). This explicit tracking avoids the need to parse lengthy textual histories to infer environment states. Figure 3 illustrates this dual-memory update: the Local Agent records the failure state, cause, and pending subtasks, while the Central Agent updates global task progress and robot availability.

Reflecting HiP-LR's hierarchy, each Local

Agent A_k^{local} maintains a local memory $\mathcal{M}_k^{\text{local}}$ for its assigned subtask group $\mathcal{G}_k^T$. It tracks the group's symbolic state, action history, and failures. By accumulating the effects of completed actions, Local Agents preserve progress, ensuring any replanning is strictly restricted to unfinished subtasks.

Concurrently, the Central Agent manages a global memory $\mathcal{M}^{\text{global}}$ tracking system-wide progress, robot availability, and inter-group constraints, which Local Agents query when requesting resources. This dual structure cleanly decouples local execution from global coordination: Local Agents operate within a compact, PDDL-grounded context, while the Central Agent seamlessly manages cross-group consistency.

4 Experiment

4.1 Experimental Setup

Datasets. We evaluate HiP-LR on two benchmarks. For general long-horizon tasks, we use **MAP-THOR** (Nayak et al., 2024b), containing 45 household tasks across 5 floor plans in the AI2-THOR simulator (Kolve et al., 2017) (225 scenarios). To analyze replanning capabilities under controlled execution failures, we construct **RECAP-THOR**, a new failure-injection benchmark derived from representative MAP-THOR errors. In MAP-THOR, failure frequency and timing depend heavily on the underlying LLM and planner, making it difficult to compare recovery behaviors under identical conditions. RECAP-THOR addresses this by designing each task to fail at a predetermined action, while utilizing the same AI2-THOR simulator and floor plans. The benchmark targets four common failure modes: an unpickable target object (T1), a receptacle that cannot accommodate all required items (T2), a robot initially holding an unrelated object (T3), and a receptacle whose state changes between planning and execution (T4). Instantiating two scenario templates across five floor plans for each failure type yields 40 task instances.

Baselines. We evaluate HiP-LR against seven multi-robot planning baselines: SMART-LLM (Kannan et al., 2023), DMRS-1D and DMRS-2D (Zhang et al., 2024), LaMMA-P (Zhang et al., 2025), LaMMA-P+ (an augmented version equipped with a naive feedback loop), H-AIM (Zeng and Li, 2026), and KGLAMP (Shek et al., 2026) (see Appendix D for baseline details). Because most baselines support a maximum of three robots, we employ a three-robot setup across all evaluations and use GPT-4o (OpenAI, 2023) as the backbone LLM.

MAP-THOR					
Method	**SR↑**	**TR↑**	**C↑**	**B↑**	**SE↓**
SMART-LLM	0.04	0.12	0.25	0.09	1.00
DMRS-1D	0.23	0.53	0.76	0.61	0.54
DMRS-2D	0.27	0.53	0.76	0.67	0.52
LaMMA-P	0.30	0.50	0.54	0.34	0.57
LaMMA-P+†	0.46	0.65	0.69	0.33	0.60
H-AIM†	0.53	0.66	0.66	0.57	0.75
KGLAMP†	0.51	0.74	0.76	0.64	0.71
HiP-LR (Ours)†	0.73	0.90	0.87	0.75	0.48

RECAP-THOR					
Method	**SR↑**	**TR↑**	**C↑**	**B↑**	**SE↓**
LaMMA-P+†	0.25	0.50	0.56	0.21	0.72
KGLAMP†	0.12	0.60	0.65	0.10	0.66
HiP-LR (Ours)†	0.47	0.84	0.83	0.58	0.68

† Approaches with failure recovery capabilities.

Table 1: Benchmark results on MAP-THOR and RECAP-THOR using a three-robot setup.

Evaluation Metrics. We categorize our evaluation metrics into two main groups: task execution and replanning efficiency. For task execution, we adopt Success Rate (SR), Transport Rate (TR), Coverage (C), and Balance (B) from MAP-THOR (Nayak et al., 2024b). Additionally, we propose Step Efficiency (SE) to quantify task completion speed in terms of execution steps. For replanning efficiency, we introduce four new metrics: Replanning Success Rate (RSR), representing the fraction of successful replans; Replan Wait (RW), measuring the total robot idle time during replanning (sec/task); Replanning Time (RT), average time to generate a new plan (sec); and Replanning Token Usage (RTU), counting the LLM tokens consumed during replanning. (See Appendix E for details)

4.2 Benchmark Performance

Table 1 summarizes the benchmark results for both MAP-THOR and RECAP-THOR. We excluded baselines without a replanning module (SMART-LLM, DMRS, LaMMA-P) and H-AIM, which relies on a pre-specified Behavior Tree for failure recovery, from the RECAP-THOR experiments.

On **MAP-THOR**, HiP-LR consistently outperforms all baselines, including pure LLM-based planners (SMART-LLM, DMRS), PDDL-based method (LaMMA-P), and failure-recovery methods (LaMMA-P+, H-AIM, KGLAMP). Specifically, it

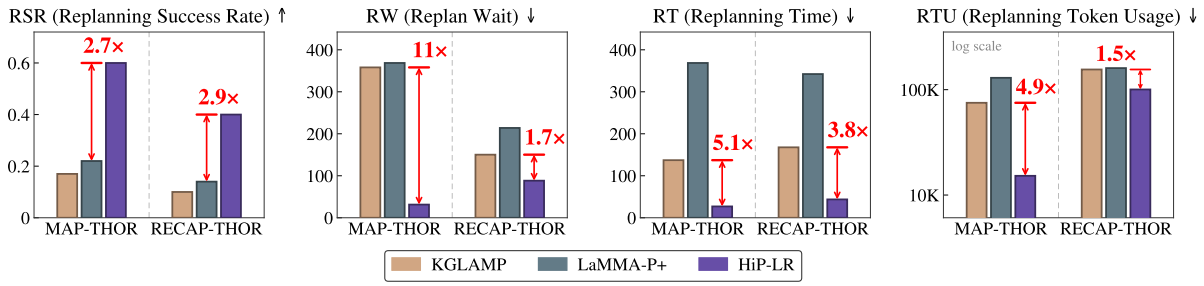

Figure 4: Failure recovery performance for HiP-LR, KGLAMP, and LaMMA-P+ on MAP-THOR and RECAP-THOR. Red arrows denote HiP-LR's improvement over the best baseline. RTU is shown on a logarithmic scale.

achieves the highest scores in task success (SR), subtask completion (TR), and object coverage (C). This demonstrates the effectiveness of our hierarchical, dependency-aware replanning for long-horizon multi-robot tasks. Furthermore, HiP-LR yields highly competitive workload balance (B) and execution efficiency (SE), proving it optimizes both task completion and parallel robot coordination.

Results on **RECAP-THOR** explicitly highlight HiP-LR's robustness under execution failures. By outperforming strong baselines like LaMMA-P+ and KGLAMP, HiP-LR validates our core hypothesis: localized replanning of only the affected subtask group is significantly more effective than global plan restarts. Even under injected failure conditions, HiP-LR successfully isolates the error to prevent failure propagation, ensuring resilient and uninterrupted execution.

4.3 Failure Recovery Analysis

Figure 4 compares the failure recovery performance on MAP-THOR and RECAP-THOR. Across both benchmarks, HiP-LR achieves the highest replanning success rate (RSR), proving its group-level strategy is more reliable than global replanning baselines. By localizing the failure scope through the dependency graph with dual-memory, HiP-LR regenerates executable plans effectively, avoiding the inefficiency of restarting the entire task plan after each failure. This consistent improvement across both the original and failure-injected benchmarks demonstrates that HiP-LR's recovery mechanism generalizes robustly beyond specific failure distributions.

In terms of replanning efficiency, HiP-LR substantially minimizes replan wait (RW), replanning time (RT), and token usage (RTU) compared to baselines. While KGLAMP and LaMMA-P+ incur severe delays from centralized or full-pipeline replanning, HiP-LR restricts replanning solely to the affected subtask group, enabling unaffected robots to seamlessly continue execution. This localized recovery design dramatically reduces both system-level idle time and LLM inference costs. Furthermore, token-usage results confirm that HiP-LR achieves highly efficient replanning by effectively constraining the context and decision space using a PDDL-based dual-memory structure.

Method	SR↑	RW↓	RSR↑	RTU↓
w/o Dual Memory	0.62	186.43	0.38	36.3k
w/o Dependency Graph	0.64	74.69	0.14	144.6k
w/o Hierarchical	0.65	57.20	0.50	37.1k
HiP-LR	0.73	34.71	0.90	11.2k

Table 2: Ablation study of HiP-LR's three core components on MAP-THOR. Highlighted cells denote each component's most affected metric.

4.4 Ablation Study

We evaluate three HiP-LR variants on MAP-THOR by independently removing one core component:

- **w/o Dual Memory.** Replaces each Local Agent's local memory with a single globally shared memory.
- **w/o Dependency Graph.** Removes the PDDL-derived graph, relying on the LLM to group subtasks without explicit causal isolation.
- **w/o Hierarchical.** Merges global and local planning into a single agent, making it equivalent to a centralized planning with failure recovery.

Table 2 summarizes the ablation results, revealing each component's contribution. Removing **Dual Memory** primarily increases replan wait, highlighting the importance of separating local histories from global memory to prevent unnecessary synchronization. Removing the **Dependency Graph** severely degrades replanning success rate and token efficiency, proving explicit causal struc-

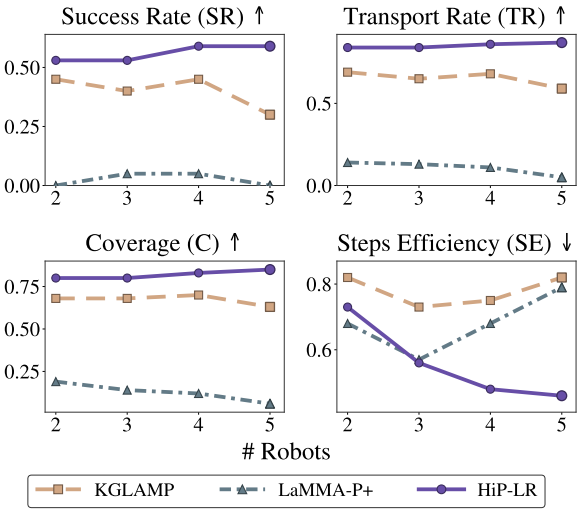

Figure 5: Scalability comparison of HiP-LR, KGLAMP, and LaMMA-P+ on MAP-THOR as the number of robots increases from two to five.

tures are vital for identifying the correct replanning scope and avoiding redundant context.

Removing the **Hierarchical** structure degrades all metrics, confirming that distributed Local Agents are vital for robust recovery. However, this variant still achieves a success rate of 0.65 (SR), outperforming all baselines in Table 1 despite acting like a centralized planner. This proves HiP-LR's superiority stems not only from its hierarchy, but largely from its dependency-graph replanning and PDDL memory, which inherently preserve symbolic structure and ensure reliable recovery.

4.5 Scalability Analysis

We evaluate scalability on the 20 largest MAP-THOR scenarios by floor area. To focus on high-level planning scalability, we utilize large scenes where introducing five robots does not trigger severe physical congestion or low-level navigation bottlenecks. Figure 5 presents the scalability results as the number of robots grows from two to five. HiP-LR maintains stable or improved success rate, transport rate, and coverage while enhancing step efficiency. This indicates HiP-LR effectively exploits additional robots for parallel execution without degrading planning quality.

Conversely, baseline methods become increasingly unstable as the team grows. KGLAMP and LaMMA-P+ drop notably in task success and efficiency, demonstrating that centralized replanning scales poorly as joint state spaces expand. These results highlight the limitations of centralized planning under multi-robot complexity.

Config	Tasks	Subtasks	Groups	Mean	90% CI (%)
1	18	51	2.11	-0.010	$[-1.7, +0.3]$
2	14	65	3.86	$+0.001$	$[-1.1, +0.7]$
3	10	37	2.40	-0.021	$[-3.3, -1.5]$
4	14	28	4.00	-0.015	$[-2.9, -1.2]$
All	49	181	3.09	-0.010	$[-1.6, -0.6]$

Table 3: Subtask group isolation analysis. Config: MAP-THOR scenario set. Tasks/Subtasks: total counts. Groups: average number of generated groups per task. Mean: average relative execution-time difference between parallel and isolated execution. 90% CI: confidence interval.

4.6 Group Isolation Analysis

To verify whether the proposed method produces execution-isolated subtask groups, we conduct an equivalence-based reliability analysis on MAP-THOR. For each generated group, we compare its completion time during parallel execution with other groups against its isolated execution. We compute the relative execution-time difference and evaluate if it falls within a predefined equivalence margin. Statistical equivalence between these modes indicates the group is minimally affected by others, confirming effective isolation.

Table 3 reports the results across MAP-THOR configurations. The mean relative time differences are near zero across all scenarios. Moreover, the 90% confidence intervals remain narrow and centered around zero, with an overall interval of $[-1.6\%, -0.6\%]$. This demonstrates that parallel execution introduces no meaningful delay compared to executing a group alone, proving our strategy successfully generates independent and execution-isolated subtask groups.

5 Conclusion

We presented HiP-LR, a hierarchical framework for long-horizon multi-robot planning with localized failure recovery. Partitioning the subtask graph into PDDL-derived dependency groups enables a Central Agent to handle global allocation, while Local Agents manage detailed scheduling and recovery. Upon encountering a failure, only the affected group is replanned, leaving independent groups uninterrupted. Experiments demonstrate significant improvements in task success and efficiency, as localized recovery substantially reduces wait costs, planning time, and LLM token usage. These results establish causal isolation as an effective foundational principle for scalable multi-robot planning.

6 Limitations

While HiP-LR advances multi-robot coordination, it currently relies on a fully defined initial state s_0, assuming an oracle-level, fully observable environment. In reality, robots operate under partial observability and rely on egocentric perception, making exhaustive scene-level information unattainable without active exploration. Although baseline studies also operate under this assumption of a fully observable environment, relying on externally supplied symbolic states remains a critical bottleneck for full autonomy. To bridge this gap, future work will integrate Vision-Language Models (VLMs) and real-time spatial mapping to continuously extract and update symbolic states from egocentric observations, ultimately eliminating the need for oracle assumptions.

References

Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins Sri, Anthony Barrett, Dave Christianson, and 1 others. 1998. Pddl—the planning domain definition language. *Technical Report, Tech. Rep.*

David Bai, Ishika Singh, David Traum, and Jesse Thomason. 2024. Twostep: Multi-agent task planning using classical planners and large language models. *arXiv preprint arXiv:2403.17246*.

Suraj Borate, Bhavish Rai B, Vipul Pardeshi, and Madhu Vadali. 2025. Comuros: A LLM-driven hierarchical architecture for adaptive multi-robot collaboration. In *IROS 2025 Workshop - LeaPRiDE: Learning, Planning, and Reasoning in Dynamic Environments*.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Ilche Georgievski and Marco Aiello. 2015. Htn planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156.

Rohan Gupta, Trevor Asbery, Zain Merchant, Abrar Anwar, and Jesse Thomason. 2025. Robotfleet: An open-source framework for centralized multi-robot task planning. *arXiv preprint arXiv:2510.10379*.

Malte Helmert. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

brian ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T Toshev, Vincent Vanhoucke, and 26 others. 2023. Do as i can, not as i say: Grounding language in robotic affordances. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pages 287–318. PMLR.

Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. 2023. Smart-llm: Smart multi-agent robot task planning using large language models. *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12140–12147.

Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, and 1 others. 2017. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*.

Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15339–15353.

Kehui Liu, Zixin Tang, Dong Wang, Zhigang Wang, Xuelong Li, and Bin Zhao. 2025. Coherent: Collaboration of heterogeneous multi-robot system with large language models. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10208–10214.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

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

Siddharth Nayak, Adelmo M Orozco, Marina T Have, Vittal Thirumalai, Jackson Zhang, Darren Chen, Aditya Kapoor, Eric Robinson, Karthik Gopalakrishnan, James Harrison, and 1 others. 2024a. Long-horizon planning for multi-agent robots in partially observable environments. *Advances in Neural Information Processing Systems*, 37:67929–67967.

Siddharth Nayak, Adelmo Morrison Orozco, Marina Ten Have, Vittal Thirumalai, Jackson Zhang, Darren Chen, Aditya Kapoor, Eric Robinson, Karthik Gopalakrishnan, James Harrison, and 1 others. 2024b. Map-thor: Benchmarking long-horizon multi-agent

planning frameworks in partially observable environments. In *Multi-modal Foundation Model meets Embodied AI Workshop@ ICML2024*.

Kazuma Obata, Tatsuya Aoki, Takato Horii, Tadahiro Taniguchi, and Takayuki Nagai. 2024. Lip-llm: Integrating linear programming and dependency graph with large language models for multi-robot task planning. *IEEE Robotics and Automation Letters*, PP:1–8.

OpenAI. 2023. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2(5):1.

Chak Lam Shek, Faizan M Tariq, Sangjae Bae, David Isele, and Piyush Gupta. 2026. Kglamp: Knowledge graph-guided language model for adaptive multi-robot planning and replanning. *arXiv preprint arXiv:2602.04129*.

Guangyao Shi, Yuwei Wu, Vijay Kumar, and Gaurav S Sukhatme. 2025. Pip-llm: Integrating pddl-integer programming with llms for coordinating multi-robot teams using natural language. *arXiv preprint arXiv:2510.22784*.

Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B. Tenenbaum, Leslie Kaelbling, and Michael Katz. 2024. Generalized planning in pddl domains with pretrained large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(18):20256–20264.

Yejin Son, Minseo Kim, Sungwoong Kim, Seungju Han, Jian Kim, Dongju Jang, Youngjae Yu, and Chan Young Park. 2025. Subtle risks, critical failures: A framework for diagnosing physical safety of llms for embodied decision making. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 25703–25744.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 2998–3009.

Marcus Tantakoun, Christian Muise, and Xiaodan Zhu. 2025. Llms as planning formalizers: A survey for leveraging large language models to construct automated planning models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 25167–25188.

Yongdong Wang, Runze Xiao, Jun Younes Louhi Kasahara, Ryosuke Yajima, Keiji Nagatani, Atsushi Yamashita, and Hajime Asama. 2024. Dart-llm: Dependency-aware multi-robot task decomposition and execution using large language models. *arXiv preprint arXiv:2411.09022*.

Haishan Zeng and Peng Li. 2026. H-aim: Orchestrating llms, pddl, and behavior trees for hierarchical multi-robot planning. *arXiv preprint arXiv:2601.11063*.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B. Tenenbaum, Tianmin Shu, and Chuang Gan. 2024. Building cooperative embodied agents modularly with large language models. In *The Twelfth International Conference on Learning Representations*.

Xiaopan Zhang, Hao Qin, Fuquan Wang, Yue Dong, and Jiachen Li. 2025. Lamma-p: Generalizable multi-agent long-horizon task allocation and planning with lm-driven pddl planner. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE.

Zhehua Zhou, Jiayang Song, Kunpeng Yao, Zhan Shu, and Lei Ma. 2024. Isr-llm: Iterative self-refined large language model for long-horizon sequential task planning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2081–2088.

Yuxiao Zhu, Junfeng Chen, Xintong Zhang, Meng Guo, and Zhongkui Li. 2025. Dexter-llm: Dynamic and explainable coordination of multi-robot systems in unknown environments via large language models. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10182–10189.

A PDDL Specification

Domain The domain defines general rules, constraints, and actions applicable across multiple subtask instances. In formal terms, the domain specifies the lifted representation of the planning problem, including the state space S, the set of actions A, and the transition function f.

In HiP-LR, we adopt a single domain named `allactionrobot`, based on the AI2-THOR environment (Kolve et al., 2017), as the shared symbolic environment for all subtask planning instances.

- **Types**: The domain defines two top-level object classes to organize the entities in the state space S into a type hierarchy. Specifically, *robot* covers all physical robot agents (e.g., `robot1`), and *object* covers all manipulable and environmental entities (e.g., `butterknife`, `drawer1`, `countertop2`). These types constrain valid predicate arguments and action parameters.

- **Predicates**: Predicates are logical atomic formulas that describe conditions or relationships among typed entities, collectively defining state space S. We categorize predicates as follows:

  - *Unary predicates* (P_1): These describe the properties or conditions of a single entity. *Example:* `(hand-empty robot1)`, the robot `robot1` is not holding any object; `(fridge-open fridge1)`, the fridge is currently open.
  - *Binary predicates* (P_2): These represent relationships between two entities. *Example:* `(at robot1 butterknife)`, `robot1` is adjacent to the butterknife; `(at-location butterknife countertop2)`, the butterknife is located on `countertop2`.

- **Actions** (A): Actions define how the environment transitions from one state to another. Each action $a \in A$ is characterised by its preconditions pre(a) and effects eff(a), which together define the transition function $f : S \times A \to S$. For example, the action `PickupObject(robot1, butterknife, countertop2)` has:

  - pre(a): `hand-empty(robot1)`, the robot must not be holding any object.
  - eff(a): `(holding robot1 butterknife)`, the robot now holds the object.

Problem Each planning problem instance defines a grounded subtask over the shared domain, represented as a tuple $(Objects_i, Init_i, Goal_i)$ where:

- **Objects** $Objects_i$: The specific entities instantiated for the subtask (e.g., `robot1`, `butterknife`, `countertop2`, `drawer1`, `floor`).
- **Initial state** $Init_i$: A complete assignment of predicate values describing the starting configuration (e.g., `(at-location butterknife countertop2)`, `(object-close robot1 drawer1)`).
- **Goal conditions** $Goal_i$: A set of predicate conditions that must hold in any valid goal state (e.g., `(at-location butterknife drawer1)`, `(object-close robot1 drawer1)`, `(hand-empty robot1)`).

In summary, the domain defines the common environment through (S, A, f), while each problem grounds the domain with a specific subtask instance $(Objects_i, Init_i, Goal_i)$. An example of a full PDDL domain and problem file is provided in Appendix A.1.

A.1 Example of PDDL Domain File and PDDL Problem File

End-to-end PDDL workflow. The *Domain file* (top-left) is the symbolic rule book: it specifies (i) abstract types `robot` and `object`; (ii) predicates such as `object-close(?r, ?x)` and `at-location(?o, ?loc)`; and (iii) actions `GoToObject` and `PickupObject`, whose preconditions and effects delimit all legal state transitions. The *Conceptual Mapping* arrow shows how these symbolic elements are reused downstream: types $\to$ objects and predicates $\to$ state literals in the grounded problem instance. During *Grounding* ($\Rightarrow$), the domain symbols are instantiated with concrete objects (`robot1`, `butterknife`, `drawer1`), a concrete initial state $Init_i$, and goal conditions $Goal_i$, yielding the *Problem file* (right). The Fast Downward planner (Helmert, 2006) then performs *Planning* ($\Downarrow$), binding domain actions to these grounded elements to generate an executable *Plan* (bottom). Overall, the figure traces the PDDL pipeline: from symbolic specification, through grounded problem instantiation, to concrete plan execution. This shows how domain knowledge is propagated and reused at each stage of subtask planning in HiP-LR.

Domain File (allactionrobot)

```
(define (domain allactionrobot)
(: types robot object)
(: predicates
(at ?r - robot ?x - object)
(holding ?r - robot ?o - object)
(at-location ?o - object ?l - object)
(hand-empty ?r - robot)
(object-open ?r - robot ?x - object)
(object-close ?r - robot ?x - object)
(is-fridge ?x - object)
(fridge-open ?f - object)
...)
(: action GoToObject
:parameters (?r - robot ?dest - object)
:precondition (not (inaction ?r))
:effect (and (at ?r ?dest)
(forall (?x - object)
(when (at ?r ?x)
(not (at ?r ?x))))))
(:action PickupObject
:pre (at ?r ?o)(hand-empty ?r)...
:eff (holding ?r ?o)...)
(:action PutObject ...)
(:action OpenObject ...)
(:action CloseObject ...)
...; 15 actions total
```

$\Rightarrow$
Grounding

Problem File (T_1: *Put ButterKnife in Drawer1*)

```
(define (problem put-butterknife)
(:domain allactionrobot)
(: objects
robot1      - robot
butterknife - object
countertop2 - object
drawer1     - object
floor       - object)
(: init
(not (inaction robot1))
(at robot1 countertop2)
(hand-empty robot1)
(at-location butterknife countertop2)
(at-location drawer1 floor)
(object-close robot1 drawer1))
(: goal (and
(at-location butterknife drawer1)
(object-close robot1 drawer1)
(hand-empty robot1))))
```

$\Downarrow$ *Fast Downward Planner*

Plan (6 steps, total-cost = 6)

```
GoToObject(robot1, butterknife) → PickupObject(robot1, butterknife, countertop2) →
GoToObject(robot1, drawer1) → OpenObject(robot1, drawer1) →
PutObject(robot1, butterknife, drawer1) → CloseObject(robot1, drawer1)
```

Figure 6: **End-to-end PDDL workflow for HiP-LR.** The *Domain file* (left) is the symbolic rule book: it specifies types `robot` and `object`, predicates such as `(at-location ?o ?loc)` and `(object-close ?r ?x)`, and 15 primitive action schemas. During *Grounding* ($\Rightarrow$), domain symbols are instantiated with concrete objects (`robot1`, `butterknife`, `drawer1`), an initial state, and goal conditions $Goal_i$, yielding the *Problem file* (right). The Fast Downward planner performs *Planning* ($\Downarrow$), binding domain actions to the grounded elements to produce the 6-step *Plan* that achieves $Goal_i$.

A.2 Example of Execution Logs

Example: Successful Subtask Execution (T_1: Put the ButterKnife in Drawer1)

Natural Language Instruction: Put the ButterKnife in Drawer1
PDDL Problem (generated by Central Agent):

```
(define (problem put-butterknife)
(:domain allactionrobot)
(:objects
robot1 - robot   butterknife - object
countertop2 - object   drawer1 - object   floor - object)
(:init
(not (inaction robot1))   (at robot1 countertop2)
(hand-empty robot1)
(at-location butterknife countertop2)
(at-location drawer1 floor)   (object-close robot1 drawer1))
(:goal (and
(at-location butterknife drawer1)
(object-close robot1 drawer1)   (hand-empty robot1))))
```

Assigned Robot: Robot 1 **Group**: $\mathcal{G}_1^T = \{T_1, T_2\}$
Completed Actions (logged in $\mathcal{M}_1^{\text{local}}$):

```
GoToObject(robot1, butterknife)
PickupObject(robot1, butterknife, countertop2)
GoToObject(robot1, drawer1)
OpenObject(robot1, drawer1)
DropHandObject(robot1, butterknife, drawer1)
CloseObject(robot1, drawer1)
```

Execution Result: **SUCCESS** **Replan version**: 0

Example: Failed Subtask with Failure Report (T_5: Turn on the Faucet)

Natural Language Instruction: Turn on the Faucet
PDDL Problem (generated by Central Agent):

```
(define (problem switch-on-faucet)
(:domain allactionrobot)
(:objects
robot1 - robot   faucet - object   floor - object)
(:init
(not (inaction robot1))   (at robot1 floor)
(hand-empty robot1)
(at-location faucet floor)   (is-faucet faucet))
(:goal (and
(switch-on robot1 faucet)   (faucet-on))))
```

Assigned Robot: Robot 1 **Group**: $\mathcal{G}_4^T = \{T_5\}$
Failure Cause: Faucet object is unreachable in the current environment configuration; no actions were completed.
Failure Report F_k (compiled by Local Agent for $\mathcal{G}_4^T$):

T_{fail} = T5 (switch-on-faucet)
r_{cause} = faucet object unreachable
H_{action} = [ ] (no actions completed)
$\mathcal{T}_{\text{remain}}$ = {T5}
s_k^{curr} = s_0 (unchanged; no effects applied)

Causal Isolation: $T_5 \in \mathcal{G}_4^T$ (singleton group). The failure is fully contained within $\mathcal{G}_4^T$; groups $\mathcal{G}_1^T = \{T_1, T_2\}$, $\mathcal{G}_2^T = \{T_3\}$, and $\mathcal{G}_3^T = \{T_4\}$ complete their subtasks without interruption.
Execution Result: **FAILED** **Replan triggered for**: $\mathcal{G}_4^T$ only

Figure 7: Execution log for T_1 (successful) and T_5 (failed) with failure report F_k.

B Dependency Graph Construction

The key claim of HiP-LR is that failure impact can be *structurally bounded*: when a subtask fails, only the subtasks that are causally or resource-connected to it need to be replanned. The dependency graph is the data structure that makes this containment possible. By encoding ordering constraints between subtasks as directed edges, the dependency graph lets the system identify exactly which subtasks belong to the same failure boundary, and which do not.

This appendix explains how the dependency graph is constructed from PDDL representations (Section F) and how it yields the two operational units used by HiP-LR: subtask groups for localized failure recovery, and execution layers for concurrent dispatch.

B.1 Edge Extraction

After PDDL grounding, the Central Agent constructs the dependency graph $G^T = (\mathcal{T}, \mathcal{E})$, where nodes $T_i \in \mathcal{T}$ are subtasks and directed edges $e_{ij} \in \mathcal{E}$ encode ordering constraints. An edge $T_i \rightarrow T_j$ is inserted under either of two conditions.

Causal dependency. A causal edge is inserted when a PDDL *effect* of T_i directly satisfies a necessary *precondition* of T_j. For example, if T_i picks up a pan (effect: `holding robot pan`) and T_j requires the pan to be held (precondition: `holding robot pan`), then T_j cannot begin until T_i completes, so $T_i \rightarrow T_j$. This extraction is fully deterministic from the PDDL representations; no LLM reasoning is involved.

Resource dependency. A resource edge is inserted when T_i and T_j share a manipulable object or exclusive resource (e.g., the same receptacle), creating a potential spatial or manipulation conflict if both execute concurrently. For example, two subtasks that both require opening and closing the same drawer cannot safely run in parallel. The direction of each resource edge is determined from the initial PDDL plan ordering: since each subtask's solution is a sequential action list, the relative execution order of two resource-sharing subtasks is read directly from those lists, ensuring that the inserted edge does not form a cycle. In practice, mutual resource cycles are rare in household task domains because objects have unique initial locations and subtasks consume resources in a strictly forward direction.

B.2 Structural Units Derived from the Dependency Graph

Once all edges are inserted, the dependency graph naturally yields two structural units that govern HiP-LR's execution strategy.

Subtask group $\mathcal{G}_k^T$. Each *weakly connected component* of G^T defines a subtask group. Because no directed edge crosses component boundaries by definition, a failure within $\mathcal{G}_k^T$ cannot propagate to any $\mathcal{G}_{k'}^T \neq \mathcal{G}_k^T$. This is the formal guarantee of *causal isolation*: the Local Agent assigned to $\mathcal{G}_k^T$ may suspend and replan without affecting any other group. This property holds in the worked example below.

Execution layer L_ℓ. All subtasks at topological depth ℓ in G^T (those whose longest dependency path from any source node has length exactly ℓ) form an execution layer. Subtasks within L_ℓ share no mutual precedence constraints and are dispatched concurrently across available robots. Layers are executed in increasing depth order: L_0 first (source nodes with no predecessors), then L_1 once all L_0 subtasks succeed, and so on.

Action-level Dependency Graph for local scheduling. The subtask dependency graph defines the ordering of subtasks, but robot execution operates at the level of individual primitive actions. Upon receiving subtask group $\mathcal{G}_k^T$, the Local Agent expands $\mathcal{G}_k^T$ into a low-level *action dependency graph* $\mathcal{G}_k^A$. Each subtask node T_i is replaced by the linear chain of primitive actions $\langle a_1^i, \ldots, a_{n_i}^i \rangle$ produced by Fast Downward, preserving intra-subtask ordering. Cross-subtask dependency edges from $\mathcal{G}_k^T$ are carried over by connecting the final action of T_i to the first action of each successor T_j. The Local Agent then schedules and monitors execution at this action-level granularity: a primitive action is dispatched as soon as all of its predecessors in $\mathcal{G}_k^A$ have completed successfully. This allows robots assigned to subtasks within the same execution layer to proceed concurrently wherever their action chains do not share dependencies.

B.3 Worked Example

The following example traces the full pipeline from a natural-language instruction to groups, layers, and failure containment.

Instruction: *"Organize the kitchen: put the ButterKnife and Fork in Drawer1, place the Bowl and Mug on CounterTop2, and turn on the Faucet."*

Step 1: Subtask identification. The Central Agent decomposes the instruction into five subtasks:

- T_1: Put ButterKnife in Drawer1
- T_2: Put Fork in Drawer1
- T_3: Place Bowl on CounterTop2
- T_4: Place Mug on CounterTop2
- T_5: Turn on Faucet

Step 2: Edge extraction. PDDL grounding reveals one ordering constraint. $T_1 \rightarrow T_2$ is a *resource* dependency: both subtasks require exclusive access to `drawer1`, which must be opened and closed between successive placements; running them concurrently would cause a manipulation conflict. T_3, T_4, and T_5 share no PDDL effects, preconditions, or exclusive resources with each other or with the Drawer group, so no ordering edges connect them.

Step 3: Group and layer formation. The four weakly connected components and the resulting execution layers are:

$$\mathcal{G}_1^T = \{T_1, T_2\} \quad \text{(Drawer1 group, Robots 1, 2)}$$
$$\mathcal{G}_2^T = \{T_3\} \quad \text{(CounterTop2 group, Robot 3)}$$
$$\mathcal{G}_3^T = \{T_4\} \quad \text{(CounterTop2 group, Robot 1)}$$
$$\mathcal{G}_4^T = \{T_5\} \quad \text{(Faucet group, Robot 4)}$$

The global execution layers are $L_0{=}\{T_1, T_3, T_4, T_5\}$ (source nodes, dispatched concurrently at the start) and $L_1{=}\{T_2\}$ (dispatched only after T_1 succeeds). Within $\mathcal{G}_1^T$, Robots 1 and 2 serialize access to `drawer1` through a four-phase protocol (open $\rightarrow$ drop $\rightarrow$ close $\rightarrow$ repeat). Robots 3 and 4 independently execute T_3 and T_5 in parallel during L_0; Robot 1 proceeds to T_4 upon completing T_1.

Step 4: Failure and causal isolation. During execution, T_5 fails because the faucet is unreachable. Because T_5 is the sole member of singleton group $\mathcal{G}_4^T$, causal isolation contains the failure entirely within $\mathcal{G}_4^T$: the Local Agent for $\mathcal{G}_1^T$ completes T_1 and T_2 (ButterKnife and Fork stored in Drawer1), the Local Agent for $\mathcal{G}_2^T$ completes T_3 (Bowl placed on CounterTop2), and the Local Agent for $\mathcal{G}_3^T$ completes T_4 (Mug placed on CounterTop2), all without interruption. Only the Local Agent for $\mathcal{G}_4^T$ initiates recovery.

C RECAP-THOR: A Benchmark for Replanning Capability under Failures

C.1 Motivation

While MAP-THOR (Nayak et al., 2024b) evaluates long-horizon multi-agent tasks, the rarity of naturally occurring execution failures limits its utility for assessing replanning capabilities. To address this, we introduce RECAP-THOR (REplanning CAPability under failures in THOR), a failure-injection benchmark where every task embeds a controlled failure specifically designed to trigger replanning.

The four failure categories in RECAP-THOR are empirically grounded. Based on execution logs from preliminary MAP-THOR experiments, we identified four primary failure types: unpickable objects (T1), lack of valid placement positions (T2), single-hand capacity limits (T3), and state mismatches in receptacles (T4). While sharing the AI2-THOR simulator and floorplans with MAP-THOR, RECAP-THOR utilizes entirely new, non-overlapping task instructions tailored to evaluate these specific failure conditions.

C.2 Failure Types and Injection Mechanisms

RECAP-THOR includes four failure types, each requiring a distinct replanning behavior. For each type, we describe the injection mechanism, failure condition, and the expected "correct recovery". Note that the correct recovery is descriptive: any recovery that completes the affected subtask and satisfies the task goal is counted as successful in RSR evaluation (Appendix E).

T1 (Unpickable Object). We include an object with `CanPickup=False` as part of the task goal. The first pickup action on that object fails during execution. The correct recovery is to abandon the subtask involving the unpickable object and complete the remaining subtasks.

T2 (No Valid Position). We configure the target receptacle so that it cannot hold all listed items, either by pre-filling it with non-task filler objects or by including more items than fit within its physical capacity (as determined by AI2-THOR's collision-based placement check). The first placement action on that receptacle fails. The correct recovery is to place the excess items in a different receptacle of the same type.

RECAP-THOR Task Scenarios			
Scenario	**Room Type**	**Failure Type**	**Floorplans**
Clear Countertop (Include Toaster)	Kitchen	T1	4, 7, 9, 11, 16
Tidy Livingroom (Include Houseplant)	Living	T1	201, 209, 215, 224, 230
Overfill Drawer (Utensils)	Kitchen	T2	2, 4, 5, 25, 30
Overflow Kitchen Cabinet	Kitchen	T2	2, 6, 7, 8, 30
Put Ingredients on Counter	Kitchen	T3	1, 2, 7, 13, 16
Organize Livingroom Items	Living	T3	201, 209, 215, 224, 230
Organize Kitchen (Cabinet Closed)	Kitchen	T4	2, 5, 16, 20, 30
Organize Kitchen (Drawer Closed)	Kitchen	T4	2, 5, 16, 20, 30

Table 4: RECAP-THOR task scenarios. Each scenario belongs to one of the four failure types, and is applied to five floorplans for a total of 40 instances. Floorplan numbers follow AI2-THOR's convention: 1–30 for kitchens and 201–230 for living rooms.

T3 (One-Hand Limit). We place a non-target object in the robot's hand at episode start. The first pickup action fails because the hand is already occupied. The correct recovery is to empty the hand and then retry the pickup.

T4 (State Mismatch). Before planning, we open the target receptacle so that the planner observes it as open. After planning completes but before execution begins, we close the receptacle. The first placement action targeting that receptacle then fails. The correct recovery is to insert an `Open` action for the receptacle.

C.3 Task Instances

We design 8 scenario templates covering the four failure types (two templates per type). By deploying these templates across 5 distinct floorplans, we generate a total of 40 unique task instances. During evaluation, each instance corresponds to a single episode executed by a team of three robots. Table 4 provides the complete list of scenarios.

D Baseline Descriptions

We evaluate HiP-LR against seven baselines spanning four categories: (i) no symbolic representation and no recovery (SMART-LLM, DMRS); (ii) PDDL grounding without recovery (LaMMA-P); (iii) naive feedback loops with global restarts (LaMMA-P+); and (iv) recovery at an inappropriate granularity (H-AIM, KGLAMP).

SMART-LLM (Kannan et al., 2023). This pure LLM-based planner decomposes tasks and assigns subtasks to robots via skill-based matching. It lacks both symbolic representations and failure recovery mechanisms, serving as the lower bound in our comparative evaluation.

DMRS-1D and DMRS-2D (Zhang et al., 2024). These are decentralized CoELA planners where robots coordinate through inter-robot dialogue to reduce information asymmetry. **DMRS-1D** utilizes a single round of dialogue, while **DMRS-2D** extends this to two rounds. CoELA provides neither symbolic grounding nor failure recovery.

LaMMA-P (Zhang et al., 2025). As the closest symbolic predecessor to HiP-LR, this baseline integrates PDDL planning with LLMs for multi-robot task allocation. However, it lacks a dependency graph and provides no failure recovery; consequently, any execution failure brings the entire system to a complete halt.

LaMMA-P+. An augmented version of LaMMA-P equipped with a naive feedback loop. Upon encountering a failure, it fetches updated environment metadata from the simulator and entirely reruns its six-module planning pipeline from the current state, effectively executing a costly global restart.

H-AIM (Zeng and Li, 2026). This framework supports failure recovery, but strictly at the *action level* via Behavior Trees. When a node fails, a local subtree attempts to retry or substitute the action. Crucially, it cannot propagate updated world states to downstream subtasks if a failure invalidates their preconditions.

KGLAMP (Shek et al., 2026). Serving as our strongest replanning baseline, KGLAMP utilizes a knowledge graph to track the environment state and regenerates PDDL plans via the Fast Downward solver upon failure. However, its replanning is entirely *centralized*, forcing all robots to idle until a new global plan is issued.

E Metric Details

We adopt four standard task metrics (SR, TR, C, B) from MAP-THOR (Nayak et al., 2024b). To evaluate our specific setting, we additionally propose the Step Efficiency (SE) for measuring parallel execution efficiency, alongside four novel replanning metrics (RSR, RW, RT, RTU). Throughout the evaluation, subtask completion and action execution are verified using AI2-THOR's object state metadata, which confirms whether the simulator state satisfies the predefined goal conditions.

An **episode** is defined as a single execution of instruction $\mathcal{I}$ on a fixed floorplan. All metrics are computed per episode and averaged over E episodes in total.

Task Metrics.

- **Success Rate (SR)** [↑, $[0, 1]$]: The fraction of episodes where every subtask is completed successfully.
- **Transport Rate (TR)** [↑, $[0, 1]$]: The fraction of subtasks completed per episode, providing a finer-grained measure of overall task progress.
- **Coverage (C)** [↑, $[0, 1]$]: The fraction of ground-truth target objects that robots interact with at least once per episode, measuring the system's ability to correctly identify implicitly specified objects.
- **Balance (B)** [↑, $[0, 1]$]: A measure of how uniformly the workload is allocated across robots, averaged over E episodes.
- **Step Efficiency (SE)** [↓, $[1/|\alpha_{seq}|, 1]$]: A novel metric quantifies how fast a task is completed in terms of execution steps. For each episode $e \in E$, it generates a single sequential action plan and computes the ratio of the actual execution steps to the total steps of this plan:

$$\text{SE} = \frac{1}{E}\sum_{e=1}^{E}\frac{|\alpha^e_{\text{exe}}|}{|\alpha^e_{\text{seq}}|} \tag{1}$$

  where $|\alpha^e_{\text{seq}}|$ is the total number of sequential actions across all robots in episode e, while $|\alpha^e_{\text{exe}}|$ is the number of real execution steps.

Replanning Metrics. All replanning metrics are reported as per-episode averages: values are first aggregated within an episode, then averaged across all E episodes. A replanning begins when the local agent detects a failure and ends when a new plan is dispatched to the affected robots.

- **Replanning Success Rate (RSR)** [↑, $[0, 1]$]: Within an episode, the fraction of replanning subtasks that succeed. A replanning result is considered successful if the regenerated plan executes to completion and the originally failed subtask reaches its goal condition:

$$\text{RSR} = \frac{\bar{T}^e_{success}}{\bar{T}^e_{total}} \tag{2}$$

  where $\bar{T}^e_{success}$ denote the number of successful replannings and $\bar{T}^e_{total}$ the total in episode e

- **Replan Wait (RW)** [↓, robot-seconds]: Within an episode, the cumulative robot idle time during replanning, measured in robot-seconds:

$$\text{RW} = \frac{1}{E}\sum_{e=1}^{E}|\mathcal{R}^{\text{halt},e}| \cdot time^e \tag{3}$$

  where $\mathcal{R}^{\text{halt},e} \subseteq \mathcal{R}$ is the set of halted robots and $time^e$ the duration of the replannings in episode e.

- **Replanning Time (RT)** [↓, seconds]: Within an episode, the average wall-clock time required to generate a new plan during replannings. The final RT is averaged over E episodes:

$$\text{RT} = \frac{1}{E}\sum_{e=1}^{E} w^e \tag{4}$$

  where w^e is the wall-clock time for the replannings in episode e.

- **Replanning Token Usage (RTU)** [↓, tokens]: Within an episode, the total number of LLM tokens consumed during replannings. The final RTU is averaged over E episodes:

$$\text{RTU} = \frac{1}{E}\sum_{e=1}^{E}(token^e_{\text{in}} + token^e_{\text{out}}) \tag{5}$$

  where $token^e_{\text{in}}$ and $token^e_{\text{out}}$ are the input and output token counts for the replannings in episode e.

F Additional Experiments

F.1 Effect of Different LLMs

Setup. To verify that HiP-LR's advantages generalize beyond GPT-4o (OpenAI, 2023), we evaluate

LLM	Method	SR↑	TR↑	C↑	B↑	SE↓	RSR↑	RT↓	RTU↓
MAP-THOR									
Llama 3.3 70B	KGLAMP	0.28	0.55	0.62	0.74	1.00	0.05	378.62s	136,760
	HiP-LR	0.53	0.77	0.79	0.51	0.58	0.40	83.96s	23,913
Qwen 2.5 72B	KGLAMP	0.21	0.42	0.44	0.15	0.52	0.10	78.65s	40,352
	HiP-LR	0.64	0.87	0.83	0.17	0.74	0.92	8.07s	7,947
RECAP-THOR									
Llama 3.3 70B	KGLAMP	0.12	0.25	0.31	1.00	1.00	0.00	–	–
	HiP-LR	0.37	0.70	0.84	0.57	0.51	0.81	201.05s	123,643
Qwen 2.5 72B	KGLAMP	0.12	0.57	0.77	0.20	0.89	0.08	370.30s	111,683
	HiP-LR	0.25	0.69	0.73	0.41	0.44	0.76	61.47s	39,723

Table 5: Effect of different LLM backbones on HiP-LR and KGLAMP. Best results in each (LLM, benchmark) group in dark beige.

both HiP-LR and our strongest baseline, KGLAMP, using two open-source models: Llama 3.3 70B and Qwen 2.5 72B. Evaluations are conducted on both MAP-THOR and RECAP-THOR. All other configurations remain identical to those in Section 4.1. The open-source LLMs are served via vLLM on a single NVIDIA B200 GPU using greedy decoding, and we apply the exact same prompts across all models to ensure a fair comparison.

Results. Table 5 summarizes the performance across all three LLMs. HiP-LR consistently outperforms KGLAMP in task success and replanning success rate metrics (SR, TR, RSR), while substantially reducing planning costs across most settings. Notably, while absolute scores naturally drop for both methods when switching from GPT-4o to the less capable open-source models, the relative performance gap in favor of HiP-LR is robustly preserved.

F.2 Analysis by Failure Type

Motivation. The main results report overall performance on RECAP-THOR. To see which failure categories benefit most from HiP-LR's localized replanning, we break down results by the four failure types (T1 to T4) defined in Appendix C.

Results. Table 6 details the performance across each failure type, demonstrating that HiP-LR consistently outperforms KGLAMP across all four categories. This advantage is most pronounced in T1 (unpickable object) and T2 (no valid position), as unaffected groups can seamlessly continue parallel execution while only the disrupted group undergoes replanning. Conversely, the performance margin narrows in T4 (state mismatch). Recovering from this specific failure merely requires inserting an `Open` action for a single receptacle, a minor adjustment that KGLAMP's monolithic approach also handles effectively.

F.3 Extended Ablation Study

This appendix extends Section 4.4 with the complete ablation results across all evaluation metrics (Table 7).

w/o Dual Memory. We replace HiP-LR's per-agent local memory with a single shared memory pool accessible to all Local Agents. Every action, observation, and failure log that would normally be stored locally is appended to this shared buffer, and each Local Agent retrieves the full buffer when generating its next action. The Dependency Graph and hierarchical structure are preserved: Local Agents are still spawned per dependency-isolated group, and the Global Planner still performs the initial plan decomposition. Only the memory layer is modified.

w/o Dependency Graph. We skip the dependency graph (DG) construction step entirely and let the LLM produce groups directly from the PDDL-decomposed subtasks. We use the same LLM backbone (GPT-4o). The LLM outputs a JSON list of subtask groups, and each group defines one Local Agent's scope. Group partitioning is thus performed solely by the LLM's judgment, without any explicit structure. Dual Memory and hierarchical execution are preserved.

w/o Hierarchical. We collapse the two-level architecture into a single agent that performs both

RECAP-THOR (failure type)											
Method	**Task type**	**LM**	**SR↑**	**TR↑**	**C↑**	**B↑**	**SE↓**	**RSR↑**	**RW↓**	**RT↓**	**RTU↓**
HiP-LR	T1	GPT-4o	0.50	0.84	0.50	0.34	0.71	0.29	90.56	30.06	67,059
	T2	GPT-4o	0.60	0.96	1.00	0.60	0.99	0.22	110.30	38.24	141,661
	T3	GPT-4o	0.40	0.85	0.90	0.62	0.88	0.58	88.09	34.15	108,448
	T4	GPT-4o	0.40	0.70	0.90	0.87	0.68	0.50	64.30	51.86	84,600

Table 6: Per failure-type results of HiP-LR on RECAP-THOR. Best in dark beige, second-best in light beige.

Method	**SR↑**	**TR↑**	**C↑**	**B↑**	**SE↓**	**RSR↑**	**RW↓**	**RT↓**	**RTU↓**
w/o Dual Memory	0.62	0.81	0.78	0.45	0.64	0.38	186.43	74.46	36,280
w/o Dependency Graph	0.64	0.84	0.83	0.28	0.45	0.14	74.69	74.67	144,644
w/o Hierarchical	0.65	0.81	0.81	0.70	0.49	0.50	57.20	73.69	37,059
HiP-LR (full)	0.73	0.90	0.87	0.75	0.48	0.90	34.71	26.98	11,219

Table 7: Ablation results across all evaluation metrics. Best in dark beige, second-best in light beige.

global planning and execution. This single agent maintains one memory buffer and operates over the entire plan without dependency-group partitioning. To isolate the effect of hierarchy itself, we keep the Dependency Graph computation in place but use its output only as ordering hints rather than as Local Agent boundaries.

G Qualitative Analysis

This section presents two representative qualitative scenarios that highlight behavioral differences between HiP-LR and prior methods under runtime failures.

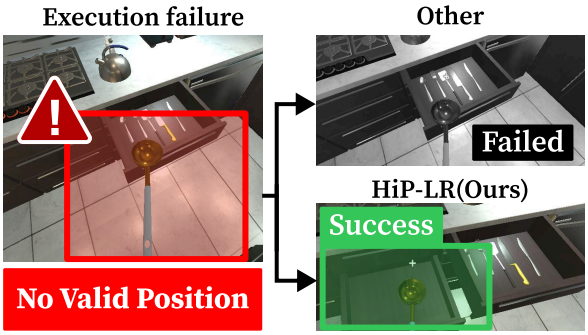

Figure 8: Example 1: Overfill Drawer. Baselines without replanning terminate when the drawer fills up, while HiP-LR redirects items to an alternative drawer.

Example 1: No Valid Position. Figure 8 illustrates a representative "Overfill Drawer" scenario (T2 failure). As shown on the left, in this long-horizon task, utensils are placed into the same drawer one or two at a time, eventually filling it and causing the next placement to fail. As shown on the right, baselines without a replanning mechanism, such as LaMMA-P, simply terminate the task at this point. HiP-LR, in contrast, detects the failure, performs localized replanning, and redirects the remaining items to an alternative drawer of the same type, successfully completing the task.

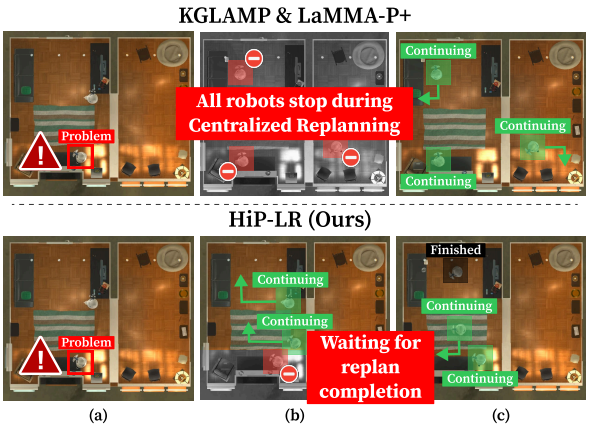

Figure 9: Example 2: One-Hand Limit. (a) A grasp failure occurs when a robot already holding a vase tries to grasp another. (b) KGLAMP halts all robots during global replanning. (c) HiP-LR halts only the affected group, letting other robots continue and finish earlier.

Example 2: One-Hand Limit. As shown in (a) of Figure 9, a robot already holding a vase in one hand attempts to grasp another vase, but the grasp fails because no hand is available. As shown in (b), centralized baselines such as KGLAMP halt *all* robots and regenerate the entire global plan, forcing even unrelated robots to idle throughout the replanning interval. In contrast, HiP-LR suspends only the robots in the affected dependency-isolated group, while robots in other groups continue executing their subtasks in parallel.

H Subtask Isolation Discrepancy

Computation of the 90% confidence interval. For configuration c, task t, and subtask i, we define the per-subtask isolation discrepancy as

$$d_{c,t,i} = \left| \frac{t^{\text{par}}_{c,t,i}}{T^{\text{par}}_{c,t}} - \frac{t^{\text{seq}}_{c,t,i}}{T^{\text{seq}}_{c,t}} \right|,$$

where $t^{\text{par}}_{c,t,i}$ is the runtime of subtask i when all subtasks of task t are executed in parallel, and $T^{\text{par}}_{c,t} = \sum_j t^{\text{par}}_{c,t,j}$ is the sum of the runtimes of all subtasks of task t under the parallel execution setting. Similarly, $t^{\text{seq}}_{c,t,i}$ is the runtime of the same subtask when executed in isolation, and $T^{\text{seq}}_{c,t} = \sum_j t^{\text{seq}}_{c,t,j}$ is the sum of the runtimes of all subtasks of task t under the isolated execution setting. Let $D_c = \{d_{c,t,i}\}$ be all per-subtask observations in configuration c, with sample size $n_c = |D_c|$, sample mean $\bar{d}_c = \frac{1}{n_c} \sum_{d \in D_c} d$, and sample standard deviation $s_c = \sqrt{\frac{1}{n_c - 1} \sum_{d \in D_c} (d - \bar{d}_c)^2}$. The reported two-sided 90% confidence interval is computed as

$$\text{CI}_{90\%}(c) = \bar{d}_c \pm t_{0.95,\, n_c - 1} \frac{s_c}{\sqrt{n_c}} \approx \bar{d}_c \pm 1.9 \frac{s_c}{\sqrt{n_c}},$$

where $t_{0.95,\, n_c - 1}$ is the 95th percentile of Student's t-distribution with $n_c - 1$ degrees of freedom. Values of $d_{c,t,i}$ close to zero indicate that each subtask preserves its relative execution share even when executed together with other subtasks.

I Baseline Settings

The following settings apply to the main experiment (Section 4.1). To ensure a fair comparison, all baselines are evaluated under identical experimental conditions. First, every method begins from the same initial environment state s_0. Second, all methods operate over the same AI2-THOR domain with an identical action set. Furthermore, all PDDL-based methods (LaMMA-P, LaMMA-P+, and KGLAMP) share the same PDDL domain used by HiP-LR. Third, GPT-4o (OpenAI, 2023) serves as the LLM backbone for all methods, with temperature and other generation hyperparameters held fixed. Where replanning is supported, the maximum number of replanning attempts per episode is also equalized to six times across baselines.

Regarding implementation, SMART-LLM (Kannan et al., 2023), LaMMA-P (Zhang et al., 2025), and H-AIM (Zeng and Li, 2026) are evaluated using their original publicly released code. KGLAMP (Shek et al., 2026) is re-implemented from scratch based on the methodology described in the original paper, as no public code was available at the time of evaluation. LaMMA-P+ is built upon the LaMMA-P codebase with a replanning module added. CoELA (DMRS-1D, DMRS-2D) (Zhang et al., 2024) is adapted from its original codebase with modifications to fit the evaluation protocol. The details of these modifications are described below.

LaMMA-P+: LaMMA-P extended with replanning

Preserved components: task decomposition module and agent prompting strategy from the original LaMMA-P.
Added component: replanning loop, triggered upon any action failure during execution.
Replanning input: when a failure is detected, the LLM receives the structured failure information below, together with live AI2-THOR environment metadata:

T_{fail} = failed subtask
r_{cause} = error message from execution
$\mathcal{T}_{\text{remain}}$ = remaining unfinished subtasks

Replanning scope: global (over all remaining subtasks $\mathcal{T}_{\text{remain}}$).

KGLAMP: Re-implemented from the original paper

Re-implemented components:

```
(1) Knowledge-graph construction module
(2) Centralized planner over the global KG
(3) Global replanning loop triggered on failure
```

Preserved components: KG construction logic, centralized planning structure, and KG-querying strategy from the original paper.
Aligned for fair comparison: To eliminate prompt-induced performance differences, the LLM prompts (planning, replanning, and KG-querying) are unified with those used by HiP-LR.
Replanning input: upon any failure, the centralized planner receives a global state update over the entire KG together with the failed action's context.
Replanning scope: global.

DMRS-1D / DMRS-2D: Decentralized dialogue baselines from CoELA

Variants:

DMRS-1D: Each robot proposes its next action via a single round of inter-agent dialogue, after which a designated robot aggregates the exchanged messages into the final action decision.
DMRS-2D: Extends DMRS-1D to two dialogue rounds, giving robots that speak first in the first round the opportunity to revise their proposals after observing later contributions, thereby reducing the gap in available context across speaking order.

Preserved components: agent architecture, inter-agent dialogue mechanism, and prompting templates from the original DMRS implementation.
Modifications: We follow the evaluation configuration (e.g., step budget) provided by the official CoELA baseline of the MAP-THOR benchmark, so that DMRS-1D and DMRS-2D are evaluated under the same protocol as all other baselines in this work.
Replanning: DMRS-1D and DMRS-2D do not include an explicit replanning mechanism beyond inter-robot dialogue; RW, RT, RSR, and RTU are not evaluated.

Figure 10: Implementation details of modified and re-implemented baselines. **LaMMA-P+**: LaMMA-P extended with a naive global replanning loop. **KGLAMP**: re-implemented from the original paper with centralized knowledge-graph-based replanning. **DMRS-1D / DMRS-2D**: decentralized CoELA variants with one and two rounds of inter-robot dialogue, respectively.

J Prompts with Examples

Prompt Module: Subtask Decomposition

Purpose. The subtask decomposition prompt is designed to decompose a natural-language task into a set of environment-grounded subtasks that an agent can directly consume. Given the runtime environment description and the robot's skill set, the LLM emits one subtask per atomic goal, each annotated with the required skills and the involved objects.

Design Principles. The prompt is organized around three invariants that the LLM must preserve:

- *Object grounding.* Every referenced object must exist in the runtime-injected scene object list.
- *Skill grounding.* Subtask actions are restricted to the available skill set, decoupling decomposition from the specific robot configuration (we use `robot[0]`'s skill list to keep the prompt robot-count agnostic).
- *Parallelism.* Whenever possible, subtasks are emitted in an independent form so that downstream stages can exploit parallelism.

Runtime Inputs. Three placeholders are substituted at runtime before the prompt is sent to the LLM.

```
{environment_objects}     scene object list with locations and rounded
                          coordinates (coordinates rounded to 2 decimals)
{available_robot_skills}  robot skill set (robot-count agnostic; uses
                          robot[0])
{task_description}        natural-language task instruction
```

Core Instruction (excerpt). We show the key part of the instruction below. The full prompt is available in our code release.

```
ENVIRONMENT OBJECTS = {environment_objects}
AVAILABLE ROBOT SKILLS = {available_robot_skills}

You MUST ground every referenced object to ENVIRONMENT OBJECTS.
You MUST use only actions whose names are in AVAILABLE ROBOT SKILLS.
The destination of each subtask MUST DIFFER from the object's current location.

# GENERAL TASK DECOMPOSITION
Decompose and parallelize subtasks wherever possible.
# Task Description: {task_description}
```

Output Schema. The LLM returns an enumerated list of subtasks. Each subtask specifies its required skills and the involved objects together with their current locations, in the following format:

```
# SubTask N: <title>
  Skills Required: <skill list>
  Related Objects: <obj(Location=...) list>
```

Example Output. For the task *"Put the bread, lettuce, and tomato in the fridge"*, the LLM produces three independent subtasks that the downstream scheduler can dispatch in parallel:

```
# SubTask 1: Move the bread into the fridge
Skills Required: [GoToObject, PickupObject, PutObject]
Related Objects: bread (Location=countertop1), fridge (Location=floor)

# SubTask 2: Move the lettuce into the fridge
Skills Required: [GoToObject, PickupObject, PutObject]
Related Objects: lettuce (Location=countertop2), fridge (Location=floor)

# SubTask 3: Move the tomato into the fridge
Skills Required: [GoToObject, PickupObject, PutObject]
Related Objects: tomato (Location=countertop2), fridge (Location=floor)
```

Figure 11: Subtask decomposition prompt module.

Prompt Module: Precondition Analysis

Purpose. The precondition analysis stage analyzes each subtask produced by the decomposition stage and identifies, based on the PDDL domain, the preconditions and effects required to later construct a PDDL problem instance. This stage does not itself emit a PDDL problem; rather, it produces the domain-grounded analysis that the downstream problem-construction stage consumes.

Design Principles. The prompt is organized around three invariants that the LLM must preserve:

- *Two-tier object scope.* The LLM prioritizes objects pre-selected for this subtask, but may also consider other objects from the current environment when reasoning about preconditions and effects.
- *Skill restriction.* The analysis is restricted to the subtask's required skills, drawn from the available skill set, ensuring that only feasible actions are considered.
- *Domain consistency.* The LLM may only use predicates already defined in the provided PDDL domain when describing preconditions and effects, and cannot introduce new ones.

Runtime Inputs. Seven placeholders are substituted at runtime before the prompt is sent to the LLM.

```
{environment_objects}            scene objects with rounded coordinates
{subtask_objects}                subtask-relevant objects from decomposition
{available_robot_skills}         full set of available skills
{subtask_skills}                 skills required for this subtask
{domain_content}                 PDDL domain definition (allactionrobot.pddl)
{few_shot_precondition_example}  few-shot output example
{subtask}                        subtask to analyze (dict)
```

Core Instruction (excerpt). The instruction block scopes the LLM to the current environment, the selected objects and skills, and the provided PDDL domain. The full prompt is available in our code release.

```
You are a precondition/effect analyzer for robot subtasks.
Analyze the given subtask and identify the preconditions and effects required to execute it,
using the predicates defined in the PDDL domain.

CURRENT ENVIRONMENT OBJECT LIST = {environment_objects}
SUBTASK OBJECTS = {subtask_objects}
AVAILABLE ROBOT SKILLS = {available_robot_skills}
SUBTASK SKILLS = {subtask_skills}
DOMAIN = {domain_content}

You MUST prioritize SUBTASK OBJECTS; other objects may be considered only when needed.
The analysis MUST be restricted to SUBTASK SKILLS.
All predicates MUST follow DOMAIN; do not introduce new ones.

=== SUBTASK TO ANALYZE ===
{subtask}
```

Output Schema. For each subtask, the LLM emits the subtask header, a precondition/effect block per required action, and an overall goal condition, all expressed in domain predicates. The example below illustrates the format.

Example Output. For the subtask *"Move the bread into the fridge"*, the analysis produces:

```
# SubTask 1: Move the bread into the fridge
Skills Required: [GoToObject, PickupObject, OpenObject, PutObject, CloseObject]
Related Objects: bread (Location=countertop1), fridge (Location=floor)

PickupObject(robot1, bread)
Preconditions: (at robot1 bread) (hand-empty robot1)
Effects: (holding robot1 bread) (not (hand-empty robot1))

OpenObject(robot1, fridge)
Preconditions: (at robot1 fridge) (not (open fridge))
Effects: (open fridge)

... (GoToObject, PutObject, CloseObject omitted)

Goal condition: (inside bread fridge)
```

Figure 12: Precondition analysis prompt module.

Prompt Module: PDDL Problem Generation

Purpose. The PDDL problem generation stage converts the action-level analysis from the upstream precondition stage into a valid PDDL problem file. The LLM takes the precondition/effect analysis as input and emits a single `(define (problem ...))` file that a classical PDDL planner can directly consume.

Design Principles. The prompt is organized around four invariants that the LLM must preserve:

- *Object closure.* The `(:objects ...)` block may only include objects that appear in the current environment list, preventing the planner from referencing nonexistent entities.
- *Initial-state completeness.* The `(:init ...)` block must contain all facts required to make the plan executable, drawing predicates exclusively from the PDDL domain.
- *Subtask-scoped goal.* The `(:goal ...)` block represents completion of the current subtask only, so the resulting plan does not over-reach into other subtasks.
- *Format strictness.* The output is restricted to a single PDDL problem file with no markdown or explanatory text, so that the downstream planner can parse it directly.

Runtime Inputs. Five placeholders are substituted at runtime before the prompt is sent to the LLM.

```
{subtask_objects}                subtask-relevant objects from decomposition
{subtask_skills}                 skills required for this subtask
{domain_content}                 PDDL domain definition (allactionrobot.pddl)
{few_shot_pddl_problem_example}  few-shot PDDL problem example
{precondition_stage_output}      output of the precondition stage (actions, pre/eff, goal)
```

Core Instruction (excerpt). The instruction block scopes the LLM to the selected objects and skills, the PDDL domain, and the upstream analysis. The full prompt is available in our code release.

```
You are a PDDL problem generation expert for robot manipulation tasks.
Generate a valid PDDL problem file for the given subtask.

SUBTASK OBJECTS = {subtask_objects}
SUBTASK SKILLS = {subtask_skills}
DOMAIN = {domain_content}

(:objects ...) MUST include ONLY objects from the current environment list.
(:init ...) MUST include ALL facts required to make the plan executable.
(:goal ...) MUST represent completion of THIS subtask only.

=== SUBTASK TO SOLVE ===
{precondition_stage_output}
```

Output Schema. The LLM returns a single file following the standard PDDL problem format `(define (problem ...) (:domain ...) (:objects ...) (:init ...) (:goal ...))`, with no markdown or explanatory prose.

Example Output. Taking the precondition-stage analysis of SubTask 1 (*"Move the bread into the fridge"*) as input, the LLM produces the following PDDL problem file:

```
(define (problem move-bread-to-fridge)
(:domain allactionrobot)
(:objects
robot1 - robot
bread - object   fridge - object
countertop1 - object   floor - object)
(:init
(hand-empty robot1)
(at-location bread countertop1)
(at-location fridge floor)
(not (open fridge)))
(:goal (and
(inside bread fridge)
(not (open fridge))
(hand-empty robot1))))
```

Figure 13: PDDL problem generation prompt module.

Prompt Module: Subtask Dependency Graph Construction

Purpose. The dependency graph construction stage infers subtask-level dependencies and produces a directed acyclic graph that downstream schedulers use both to determine parallelism and execution order, and to distribute subtasks to local agents. The LLM is invoked once per task, taking a compact JSON summary of each subtask (key objects, preconditions, effects) as input. When PDDL preconditions alone are insufficient to capture semantic ordering, the overall task goal is consulted as additional context.

Design Principles. The prompt is organized around three invariants that the LLM must preserve:

- *Instance-level reasoning.* Dependencies are induced only by mutually exclusive resources (e.g., both subtasks acting on `drawer1`). Different instances of the same type (`drawer1` and `drawer2`) run in parallel, and open surfaces such as CounterTop, DiningTable, Floor, and Shelf never induce dependencies even when shared.
- *Typed edges.* Each dependency is labeled either `causal` or `resource`, so that the scheduler can interpret why two subtasks are ordered and preserve graph acyclicity.
- *Spurious-edge avoidance.* Generic robot states (e.g., "robot not in action") and shared source locations never trigger edges, preventing over-constrained graphs that destroy parallelism.

Dependency Types. The prompt induces two kinds of edges:

- *Causal dependency.* Introduced deterministically when an effect of subtask T_i satisfies a necessary precondition of T_j. The edge direction follows this logical progression directly.
- *Resource dependency.* Introduced when subtasks share manipulable objects (e.g., the same `fork`), exclusive openable instances (e.g., both `drawer1`), or exclusive robot resources. This imposes an execution order to prevent physical conflicts or inconsistent states. To preserve acyclicity, the Central Agent aligns edge directions with the initial PDDL plan.

Runtime Inputs. Four placeholders are substituted at runtime before the prompt is sent to the LLM.

```
{task_goal}                          overall task goal in natural language
{subtask_summaries}                  per-subtask summaries (key objects, preconds, effects)
{few_shot_dependency_graph_example}  few-shot example dependency graph output
{replan_context}                     additional context block inserted only on replanning
```

Output Schema. The LLM returns a single JSON object whose `dependencies` field is a list of typed edges, each specifying a source subtask ID, a target subtask ID, an edge type, and a brief justification.

```
{
"dependencies": [
{"from": <id>, "to": <id>, "type": "causal | resource | binding", "reason": "<brief>"},
...
]
}
```

Example Output. For a task involving washing a plate and then placing it on a shelf, the same plate is manipulated across two subtasks, producing two edges between the same pair: a `causal` edge enforcing that washing precedes placement, and a `binding` edge requiring the same robot to carry the plate throughout.

```
{
"dependencies": [
{"from": 1, "to": 2, "type": "causal", "reason": "Plate must be washed before placing on
shelf"},
{"from": 1, "to": 2, "type": "binding", "reason": "Same plate must be carried by same robot"}
]
}
```

Figure 14: Subtask Dependency Graph construction prompt module.

Prompt Module: Task Allocation

Purpose. The task allocation stage assigns each subtask to one robot, given the subtask dependency graph, the per-robot capabilities, and the current robot state. The LLM is invoked once per task and returns a JSON mapping from subtask ID to robot ID. The allocation must satisfy all constraints derived from the dependency graph and the robots' inventory, while optimizing for parallelism and balanced workload.

Design Principles. The prompt is organized around four invariants that the LLM must preserve:

- *Single-hand constraint.* Each robot has a single hand and may hold at most one object at any time, so two subtasks requiring different pickups within the same component must be assigned to different robots.
- *Inventory consistency.* If a robot is currently holding an object, any subtask that manipulates that object must be assigned to the same robot, and unrelated pickup-requiring subtasks should be routed elsewhere.
- *Dependency-Graph-driven parallel distribution.* Subtasks not connected in the dependency graph can run in parallel. The LLM exploits this parallelism by distributing such subtasks across robots so that all robots execute simultaneously and none remains idle.
- *No-idle-robot policy.* Every robot must receive at least one subtask. An idle robot is treated as the worst outcome, with parallelism and workload balance as secondary objectives.

Runtime Inputs. The prompt assembles six structured blocks at runtime before invoking the LLM.

```
# Robots                 mass capacity, skills, currently held object
# Subtasks               required mass, actions, distances
# Task Dependency Graph  typed dependency graph edges from the previous stage
# Execution Components   subtask groups derived from the dependency graph
# Parallel Groups        dependency graph topological levels
# Hard Constraints       holding and forced assignments (used on replan)
```

Output Schema. The LLM returns a single JSON object that maps each subtask ID (string) to a robot ID (integer), with no explanatory text or markdown.

```
{ "<subtask_id>": <robot_id>, ... }
```

Example Output. For a task with three subtasks and two robots, where subtasks 1 and 2 are independent in the dependency graph and subtask 3 depends on subtask 2, the LLM distributes the independent pair across both robots and chains subtask 3 onto the same robot as its predecessor:

```
{"1": 1, "2": 2, "3": 2}
```

Figure 15: Task allocation prompt module.