SCHEDULE-OVERCOOKED: MULTI-TASK ALLOCATION FOR LLM AGENTS UNDER MULTIPLE CONSTRAINTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) based multi-agent systems increasingly execute concurrent tasks over a shared pool of agents and tools, making the allocation of dependent subtasks a central determinant of system efficiency. Yet allocation is evaluated only through end-to-end task success, where it is confounded with low-level execution and simplified constraints. We introduce **Schedule-Overcooked**, a benchmark that fixes task decomposition and execution to isolate subtask ownership and dispatch timing. It pairs a four-level task-complexity suite with a five-level suite of composable agent, task, and resource constraints, and exposes the same allocation interface in TaskBench, Minecraft, and Factorio. Evaluating rule-based, LLM, and GNN-PPO allocators, we find that (i) the evaluated allocators vary substantially across settings, and near-full agent utilization often coexists with unfinished workflows; (ii) under compound temporal constraints, a GNN-PPO allocator attains low success already on its training configurations, and revealing capability information at evaluation time gives little benefit to the trained policy; and (iii) frozen policies transfer well between workload levels that share workflow structure but asymmetrically across constraint scenarios, and an Overcooked-trained policy remains operational on TaskBench, Minecraft, and Factorio, where limited adaptation further improves TaskBench success. Schedule-Overcooked provides a controlled testbed for developing allocators that coordinate time, resources, and uncertainty across concurrent tasks.

1 INTRODUCTION

Building multi-agent systems (MAS) that combine complementary capabilities to solve complex problems has long been a goal of artificial intelligence. Recent advances in large language models (LLMs) have brought substantial progress through system design and model training, enabling agents to collaborate on embodied tasks, tool use, and complex reasoning (Wu et al., 2023; Zhang et al., 2024a; Sun et al., 2025; Zhou et al., 2025). However, effective cooperation on an individual task does not necessarily translate into efficient execution of concurrent tasks. Practical systems must share agents and tools across tasks, coordinate dependent subtasks, and respect resource limits. A locally useful assignment can delay another task or block downstream work. This shifts the objective from individual task success to allocation efficiency across an entire workload.

We argue that three requirements are necessary to study this problem. **First, task allocation must be evaluated independently within the MAS workflow.** The complexity of environments and tasks makes optimal reference allocations difficult to obtain for direct evaluation, so existing studies assess allocation indirectly through end-to-end task success. As Observation 1 (Section 3.1) illustrates, coupling allocation with execution makes this assessment sensitive to the execution policy, hindering the separate evaluation and optimization of allocation policies.

Second, adequately modeling complex real-world constraints is essential. Existing research often simplifies the constraints governing task allocation; for example, capability-based agent selection matches subtask requirements to agent profiles (Zhou et al., 2025). In practice, allocation also depends on whether agents are busy, their execution preferences and changing success rates, concurrency limits on shared tools, and unexpected failures. Although some studies consider API

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

costs (Chen et al., 2023; Liu et al., 2025) or inference latency (Shi et al., 2026), these factors alone do not characterize task allocation in complex environments.

Third, allocation research needs benchmarks that span task domains. Kitchens, embodied construction, tool workflows, and production systems differ in their operations but share the need to allocate limited workers and resources among competing tasks. A common allocation interface with domain-specific execution adapters enables comparisons within and across these settings, exposing recurring bottlenecks and testing whether improvements extend beyond one domain.

We introduce **Schedule-Overcooked**,[1] a framework developed on Overcooked-AI (Carroll et al., 2019). Multiple cooks execute concurrent orders, while an allocator selects subtasks, owners, and dispatch times. Our contributions are:

1. **Execution-decoupled allocation evaluation.** An Assignment-to-Action Adapter executes assignments through fixed, disclosed local procedures. Four task-complexity levels increase concurrent workload, dependency complexity, and spatial demands under the same allocation interface and execution policy.
2. **Composable constraints and graded task suites.** We formalize agent, task, and resource constraints in a common specification and combine them in a cumulative five-level suite, complemented by paired tests of individual constraints.
3. **Cross-domain evaluation and diagnostic analysis.** Extensions to Minecraft, TaskBench, and Factorio share the allocation interface, enabling within-domain evaluation and cross-dataset transfer. Paired constraint relaxations and transfer analyses characterize where allocators fall short: high utilization does not imply completion, learned policies show low success already on training configurations under compound temporal constraints, and transfer is strong across workload levels that share workflow structure but asymmetric across constraint scenarios.

2 Related Work

Tool-using LLM multi-agent systems. ReAct-style agents interleave reasoning with tool calls, and conversation frameworks such as AutoGen scale this pattern to multiple roles (Yao et al., 2023; Wu et al., 2023). Subsequent systems pair a decomposer that emits a DAG with workers that execute graph nodes (Zhou et al., 2025). Their evaluations typically measure the success of an individual task; parallelism, if present, is intra-task, and cross-task sharing of idle agents is not the object of optimization.

Benchmarks for multi-task allocation. GAIA and TaskBench evaluate individual tasks (Mialon et al., 2024; Shen et al., 2024). ParaCook (Zhang et al., 2026) evaluates parallel planning, but its embodied full-action setting couples allocation with low-level execution, so the allocator cannot be scored in isolation. CostBench (Liu et al., 2025) adds tool costs and cost adaptation, covering only a slice of realistic scheduling pressure. NL⇒Schedule (Liao et al., 2026) and DynaSchedBench (Cao et al., 2026) evaluate constrained scheduling without allocation among heterogeneous embodied agents, and TimeArena (Zhang et al., 2024c) studies temporal multitasking by a single language agent. None jointly captures cross-task agent allocation, capability uncertainty, and shared-resource contention (Table 1).

Automated agent-system design. A parallel literature searches for or generates a workflow for a *single* task given a pool of agents, tools, and communication protocols. AFlow (Zhang et al., 2024b), ADAS (Hu et al., 2024), and A^{2}Flow (Zhao et al., 2025) search candidate workflows using execution feedback, while MAS-GPT (Ye et al., 2025), ScoreFlow (Wang et al., 2025b), W4S (Nie et al., 2025), ComfyGPT (Huang et al., 2025), and MAS^2 (Wang et al., 2025a) learn to generate or refine agent systems through supervision, preference optimization, or reinforcement learning. These approaches construct the workflow for an input task rather than allocate shared agents across tasks.

[1] Code: `https://anonymous.4open.science/r/Schedule-Overcooked-2BE9/`.

Table 1: Comparison of multi-task allocation benchmarks.

Benchmark	Multi-task	Agent team	Decoupled eval.	Transfer eval.	Levels	Multi-constraint
GAIA (2024)	×	×	×	×	✓	×
SciBench-MAS (2025)	×	✓	×	×	✓	×
ParaCook (2026)	✓	✓	×	×	✓	✓
TimeArena (2024c)	✓	×	×	×	✓	✓
CostBench (2025)	×	×	×	×	✓	✓
DynaSchedBench (2026)	✓	×	✓	×	✓	✓
ORS3D-60K (2025)	×	×	×	×	✓	×
CookBench (2025)	✓	×	×	×	✓	✓
AsyncHow (2024)	×	×	×	×	✓	×
WorFBench (2025)	×	×	×	✓	×	×
Robotouille (2025)	✓	✓	×	×	×	✓
Schedule-Overcooked	✓	✓	✓	✓	✓	✓

3 PRELIMINARIES

3.1 EMPIRICAL OBSERVATIONS

The following diagnostics motivate the benchmark design. Their cohorts are separate from the suite evaluations; configurations and outcomes are provided in Appendix A.

Observation 1: Existing evaluations do not decouple allocation from execution. Multi-task evaluation often couples assignments with low-level action generation, so success reflects both allocation and execution quality. In a ParaCook diagnostic, online subtask allocation with a dedicated executor raises task completion over one-shot action generation for all three tested LLMs, motivating fixed execution when comparing allocation policies (Appendix A.2).

Observation 2: Fixed heuristics and LLM allocators cover constraint combinations unevenly. Fixed-priority heuristics miss deadlines in a TaskBench diagnostic, and LLM allocators differ in their ability to coordinate dependencies and shared resources in decoupled ParaCook, even with homogeneous agents and compact workflows. This motivates evaluation across varied constraint combinations (Appendices A.4 and A.2).

Observation 3: Trained lightweight allocators handle settings with few constraints. With state-aware feasibility support, a BC-initialized PPO allocator improves completion on compact ParaCook workflows. Learned allocators perform well under basic constraints but remain limited under more complex combinations, motivating a study of how constraint difficulty affects learning (Appendix A.3 and Section 5.3).

3.2 TASK DEFINITION

We divide how multi-agent systems solve complex tasks into three stages: task decomposition, task allocation, and task execution.

Task decomposition. Following the task-graph representation of ReSo (Zhou et al., 2025), we consider N high-level tasks $\mathcal{T} = \{T_i\}_{i=1}^{N}$ and agents $\mathcal{A} = \{a_k\}_{k=1}^{M}$. Each task T_i is decomposed into a DAG $G_i = (V_i, E_i)$ whose nodes are subtasks and whose edges encode precedence. Decomposition is fixed within an evaluation.

Task allocation. At decision time t, the allocator receives an observation o_t of task progress, agent state, resources, and execution feedback. Agent, task, and resource constraints determine admissible subtask–agent assignments. The allocation policy π_θ outputs a dispatch set

$$u_t \sim \pi_\theta(\cdot \mid h_t), \qquad u_t \subseteq \mathcal{V}_t \times \mathcal{A}, \tag{1}$$

where $\mathcal{V}_t$ contains the currently exposed subtasks and h_t is the observable interaction history.

Task execution. Agents execute their assigned subtasks in a distributed manner and return execution feedback through the Assignment-to-Action Adapter (Section 4.1).

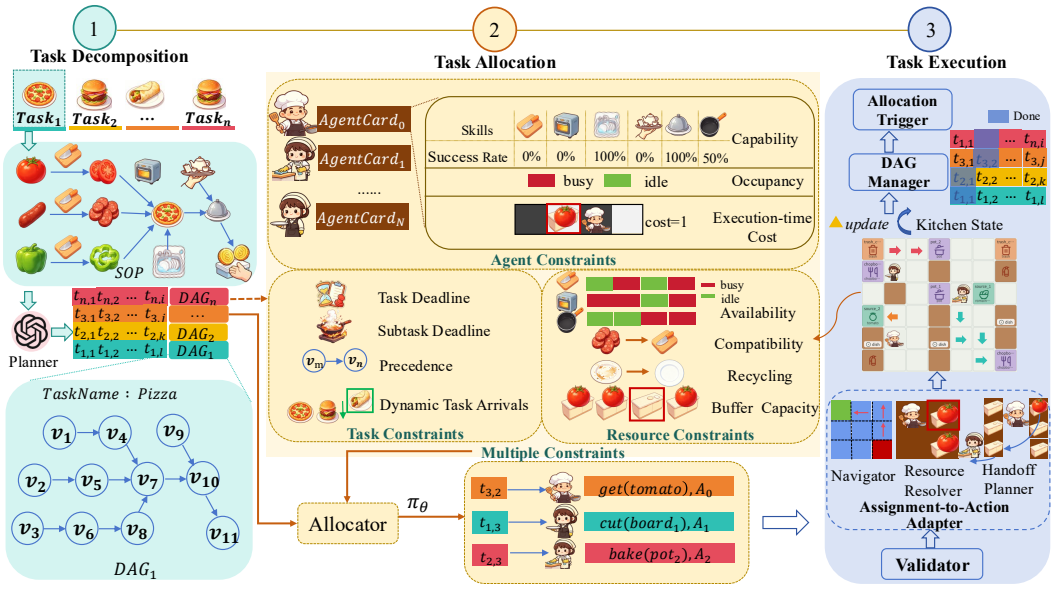


Figure 1: Overview of Schedule-Overcooked: task decomposition, allocation under agent, task, and resource constraints, and embodied execution. Allocation-only evaluation holds decomposition and execution fixed.

Allocation objective. Interacting constraints make optimal reference allocation sequences difficult to obtain, so we fix decomposition and execution and evaluate π_θ by the number of high-level tasks completed on time. Let $\mathcal{T}_H^{\text{done}}$ contain tasks completed by horizon H, with completion times C_i and deadlines d_i. The objective is

$$\max_\theta \ \mathbb{E}\Big[\textstyle\sum_{i:T_i\in\mathcal{T}_H^{\text{done}}} \mathbb{I}\{C_i \le d_i\}\Big]. \tag{2}$$

4 Schedule-Overcooked

Built on Overcooked-AI (Carroll et al., 2019), Schedule-Overcooked provides a simulator and configurable constraints for multi-task allocation. It introduces four extensions: 1) concurrent tasks with a unified, A2A-compatible environment description that represents allocation independently of the task domain; 2) three support modules that, together with the allocator, form an execution-decoupled evaluation framework; 3) agent, task, and resource constraints that model real-world requirements (Section 4.3.1); and 4) Minecraft, TaskBench, and Factorio extensions for cross-domain evaluation. Predefined configurations and customizable constraints and maps support standardized and custom evaluation.

4.1 Framework

Delegating the entire workflow to the allocator incurs repeated calls and task-state maintenance. Three auxiliary modules therefore handle decision triggering, executable-subtask management, and execution; together with the allocator they form the framework in Figure 1 (details in Appendix C).

Allocation Trigger. **Event-driven invocation.** Calling the allocator at every time step is costly and redundant, especially in real-time settings. The trigger initiates a decision when the available-subtask list changes, a new task arrives, or a fault changes execution conditions (Table 10).

DAG Manager. **Feasibility masking.** The manager tracks dependencies and execution conditions to expose admissible subtasks. **Dynamic handoff generation.** It inserts placement and transfer subtasks required by the selected assignments and task–artifact bindings.

Assignment-to-Action Adapter. **Execution decoupling.** The adapter replaces LLM action generation with a shared execution procedure, so allocation policies are compared through end-to-end

outcomes under identical execution. **Assignment-to-action conversion.** Heuristic procedures translate each assigned subtask into movement and interaction operations, plan navigation paths, and generate the agent's embodied actions.

4.2 Multi-Level Execution-Decoupled Evaluation

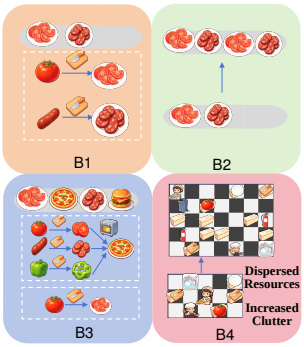


Figure 2: Four task levels.

Four task-complexity levels share the allocation interface and execution framework (Figure 2). **B1: Basic allocation.** Three recipes with isomorphic single-chain DAGs test subtask ownership and dispatch timing in compact, connected kitchens. Teams of two, three, and four agents handle 4, 5–6, and 6–10 concurrent tasks, respectively. **B2: Increased concurrency.** Task counts rise to 6, 8–10, and 12 for the same team sizes, while maps, recipes, and processing times remain fixed, focusing evaluation on distributing agent capacity. **B3: Complex dependencies.** Four or five recipes combine longer chains with two- and three-branch joins, requiring coordination of parallel branches before downstream execution. Agent counts, task counts, and maps match B2. **B4: Complex spatial scenes.** Larger layouts distribute agents and resources more sparsely, and additional kitchen entities and descriptions increase clutter, combining spatial allocation with the identification of task-relevant information. Task DAGs and functional resources match B3. **Task diversity.** Each level provides 150 configurations from ten core tasks, three terrain variants, and five tool profiles. Evaluation uses the 40 core tasks; the remaining variants support training and generalization.

4.3 Constraint-Graded Task Suites

We formalize real-world allocation requirements as agent, task, and resource constraints and combine them in a five-level suite, enabling systematic evaluation as capability, temporal, and resource demands increase.

4.3.1 Constraint Specification

An instance is $\mathcal{I} = (\mathcal{T}, \mathcal{A}, \mathcal{R}, s_0, H, \mathcal{C}, \Omega)$, with tasks $\mathcal{T}$, agents $\mathcal{A}$, resources $\mathcal{R}$, initial state s_0, horizon H, constraints $\mathcal{C} = \mathcal{C}_A \cup \mathcal{C}_T \cup \mathcal{C}_R$, and observation rule Ω. For subtask $v \in V_i$, A_v is its assigned agent set; s_v and f_v are its start and completion times.

Agent Constraints $\mathcal{C}_A$. *Capability* requires $S_v \subseteq S_a$ for each $a \in A_v$, where S_v and S_a denote required and available skills. *Occupancy* imposes $\sum_v x_{va}(t) \le b_a$, where $x_{va}(t) \in \{0,1\}$ indicates agent occupancy and b_a is capacity. *Execution-time Cost* $c_{va}^{\text{time}} \ge 0$ is the time agent a needs to execute subtask v.

Task Constraints $\mathcal{C}_T$. *Task Deadline* requires $C_i \le d_i$, where C_i is the delivery time of T_i and d_i its deadline. *Subtask Deadline* requires $f_v \le d_v^{\text{sub}}$. *Precedence* imposes $f_u \le s_v$ for each $(u,v) \in E_i$. *Dynamic Task Arrivals* require $s_v \ge r_i$ for $v \in V_i$, where r_i is the release time of T_i.

Resource Constraints $\mathcal{C}_R$. *Availability* requires $a_r(t) = 1$ for acquisition, where $a_r(t) \in \{0,1\}$ indicates that resource r is free and operational; retained outputs occupy it until removal. *Compatibility* requires $r \in \mathcal{R}_v$, the resources compatible with v. *Recycling* requires $q(t) = q(0) - N_{\text{take}}(t) + N_{\text{wash}}(t) \ge 0$, where q is clean-plate stock and N_{take}, N_{wash} count withdrawals and washed returns. *Buffer Capacity* imposes $\sum_z b_z(t) \le B_{\text{buf}}$, where $b_z(t) \in \{0,1\}$ indicates that artifact z occupies one of B_{buf} storage slots until downstream pickup.

4.3.2 Five-Level Constraint Suite

Five composite scenarios C1–C5 combine constraints so that restrictions accumulate (Table 2). Adjacent levels may change several factors at once, so level differences are not single-factor comparisons; the paired interventions in Section 5.4 relax individual constraints. **C1: Basic constraints** combines capability matching with finite resources. **C2: Stronger competition** increases workload and resource pressure with observable skill success rates. **C3: Temporal constraints** adds task deadlines and dynamic arrivals. **C4: Local expiry** adds subtask deadlines and equipment out-

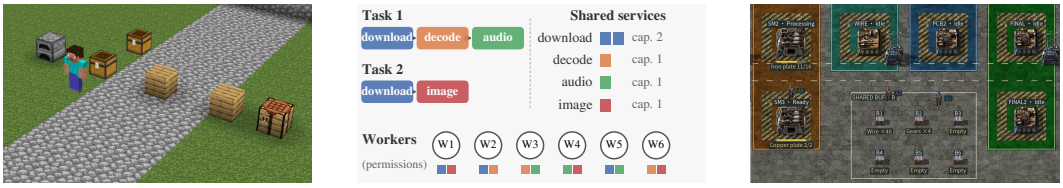
Table 2: Constraint configuration of the five-level constraint suite. Colored entries mark constraints introduced or changed at each level.

Level	Agent	Task Constraints			Resource Constraints			
	Capability visibility	Dynamic arrivals	Task deadline	Subtask deadline	Availability	Compatibility pressure	Recycling supply	Buffer supply
C1	Visible	–	–	–	Stable	Low	Ample	Ample
C2	Visible	–	–	–	Stable	**High**	**Limited**	**Limited**
C3	Visible	✓	✓	–	Stable	High	Limited	Limited
C4	Visible	✓	✓	**Local**	**Intermittent**	High	Limited	Limited
C5	**Hidden**	✓	✓	**Cascading**	Intermittent	High	**Scarce**	**Scarce**

ages. **C5: Hidden capabilities and cascading penalties** hides skill success rates and lets overdue subtasks trigger fires that disrupt subsequent work.

4.4 Cross-Domain Evaluation Support

The extensions introduce domain-specific constraints under the shared allocation interface (Figure 3); Appendix D defines their operation contracts and completion checks. **Minecraft** (VillagerBench-based; Dong et al., 2024) requires workers to coordinate limited material stocks, container access, and shared building areas across dependent crafting and construction subtasks. **TaskBench** tool workflows (Shen et al., 2024) combine typed input dependencies with worker-specific permissions and execution costs; concurrent tasks compete for services with bounded capacity and variable latency. **Factorio** production workflows (Hopkins et al., 2025) couple material balances with machine throughput and transport capacity; concurrent tasks share intermediate materials, compatible workstations, and bounded storage.



(a) Minecraft (b) TaskBench (schematic) (c) Factorio

Figure 3: Task scenes in the three extension domains.

5 Experiments

We examine how the evaluated allocators behave as workloads and constraints grow (Section 5.2), how a learned allocator trains and responds to constraint relaxations (Sections 5.3 and 5.4), and how frozen and adapted policies perform on other levels and datasets (Sections 5.5 and 5.6). Protocols are given in Appendix F.

5.1 Datasets and Baselines

The **basic suite** contains ten core tasks at each of B1–B4, varying concurrency, dependencies, and spatial complexity. The **constraint suite** contains five core families at each of C1–C5. Policies train on separate configuration variants and are selected on source validation data. **Success rate (SR)** is the case-macro fraction of high-level tasks completed within their time limits. **Agent utilization (AU)** is occupied agent-time divided by total available agent-time over the episode (Zhang et al., 2026), averaged equally over cases. All allocators share the execution interface. **Serial** permits one active subtask, including background processing; **Random-Feasible** samples a feasible assignment; **EDF** prioritizes deadlines; **Greedy-Time** prioritizes estimated execution cost; **HEFT** uses graph-based priorities (Topcuoglu et al., 2002). **GNN-PPO** learns assignments with a heterogeneous graph actor–critic (Song et al., 2023), and **LLM allocators** generate assignments from public task, agent, and resource states.

Table 3: Basic-suite allocation performance.

Method	B1		B2		B3		B4	
	SR (%)	AU (%)	SR (%)	AU (%)	SR (%)	AU (%)	SR (%)	AU (%)
Serial	45	30	28	30	34	30	32	30
Random-Feasible	80	93	31	98	15	99	10	99
EDF	**95**	90	67	95	66	96	59	98
Greedy-Time	93	92	67	96	**67**	96	59	98
HEFT	65	94	5	100	0	100	1	100
Gemini 3.8 Flash	88	93	50	100	42	100	33	100
Grok 4.6	63	99	67	100	58	100	50	100
GPT-5.6-Sol	88	91	50	100	42	100	50	100
GNN-PPO	90	86	**70**	93	66	91	**63**	90

Table 4: Multi-constraint allocation performance.

Method	C1		C2		C3		C4		C5	
	SR (%)	AU (%)	SR (%)	AU (%)	SR (%)	AU (%)	SR (%)	AU (%)	SR (%)	AU (%)
Serial	96	43	59	22	13	20	22	22	16	21
Random-Feasible	**100**	66	87	54	20	49	18	52	6	50
EDF	**100**	65	100	56	42	61	**55**	64	11	57
Greedy-Time	**100**	65	100	56	26	53	35	57	10	53
HEFT	**100**	61	52	40	21	49	10	44	3	52
Gemini 3.8 Flash	**100**	67	**100**	58	**60**	64	50	60	**28**	61
Grok 4.6	**100**	69	83	50	46	63	36	60	21	59
GPT-5.6-Sol	**100**	66	100	61	48	65	39	58	24	60
GNN-PPO	98	61	98	57	34	54	45	57	16	56

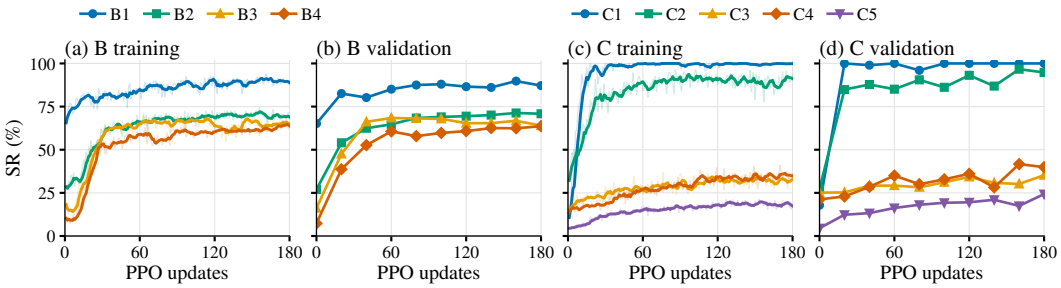


Figure 4: Learning through update 180. Faint traces are raw training means, solid curves trailing five-update means; validation is unsmoothed.

5.2 Baseline Performance

The gap between prioritized and random dispatch widens with concurrency. From B1 to B2, EDF falls from 95% to 67%, whereas Random-Feasible falls from 80% to 31% (Table 3), so the difference between them more than doubles as more tasks compete for the same agents.

High utilization does not imply high completion. HEFT and the LLM allocators keep agents occupied nearly 100% of the time on B2–B4, yet HEFT completes almost no tasks and the LLMs match or trail EDF and GNN-PPO. Occupancy includes agents that hold assignments while blocked on inputs, equipment, or storage, so committing idle agents can coexist with little progress on complete workflows (Appendix F analyzes the HEFT case).

The evaluated allocators exhibit substantial performance variation across settings. No single method is best on every level (Table 4): EDF achieves the highest success on C4 but 11% on C5; the LLM allocators achieve the highest success on C3; and GNN-PPO is strongest on B2 and B4 but not on C3–C5. On C5, which combines hidden capabilities with cascading penalties, no method exceeds 28%.

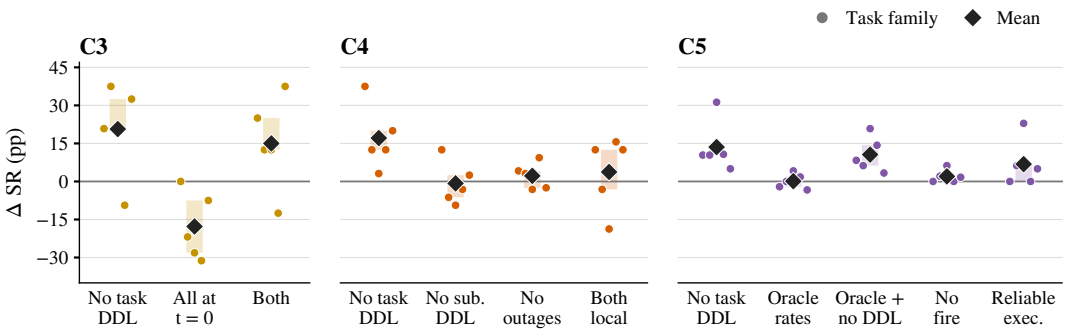


Figure 5: Paired constraint relaxations on fixed policies. Points: five task-family effects; diamonds: means; vertical bands: family interquartile ranges.

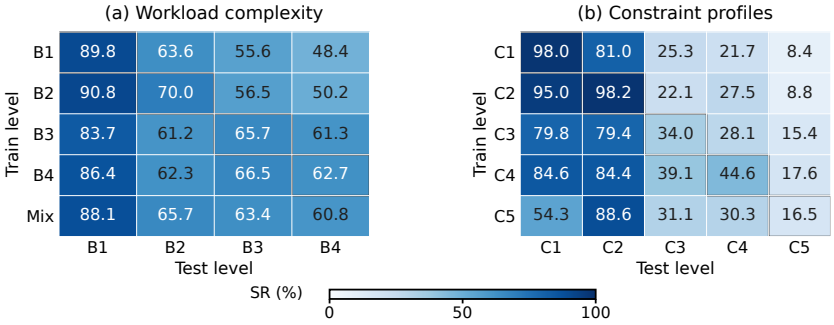


Figure 6: Frozen-policy transfer on paired core cases. Rows are training sources and columns test levels; outlines mark matched levels. Full B–C matrices are in Appendix F.

5.3 Training Performance under Compound Constraints

On B1–B4 and C1–C2, training and validation success rise together and saturate within about 60 updates (Figure 4). On C3–C5, the composite scenarios that add task deadlines and dynamic arrivals to resource competition, training success stays below 40% throughout and validation tracks it (35/40/24% at update 180). Low training success shows that the observed limitations are not confined to held-out configurations. Delayed completion rewards make temporal credit assignment a plausible contributor, but these experiments do not distinguish it from exploration, representation, training-budget, or instance-feasibility limitations.

5.4 Fixed-Policy Responses to Constraint Relaxations

We relax one constraint at a time on paired cases while holding the trained policies and horizons fixed (Figure 5). **Task deadlines.** Removing task deadlines yields the largest mean gain among the tested relaxations at every level (+14 to +21 points), whereas removing subtask deadlines or outages on C4 changes success by at most 2 points. Releasing all tasks at once lowers C3 success by 18 points. **Capability information.** Revealing success probabilities at evaluation time provides little benefit to the tested C5 policy (less than one point), whereas making capable executions reliable adds 7 points. This does not establish that capability information would be unhelpful to policies trained to exploit it. **Remaining failures.** After task deadlines are removed, success reaches 50/51/30% on C3/C4/C5. Many tasks thus remain unfinished within the fixed horizon, which may involve resource coordination, execution risk, and the remaining time limits.

5.5 Transfer across Workload and Constraint Settings

We evaluate frozen policies on other levels (Figure 6). Among workload levels, B3 and B4, which share task DAGs but differ in layout, transfer in both directions within 1.4 points of the target specialists, and a single policy trained on all B levels stays within 4.3 points of every specialist. Transfer among constraint scenarios is asymmetric: C2-to-C5 reaches 9% and C5-to-C2 reaches 89%, while

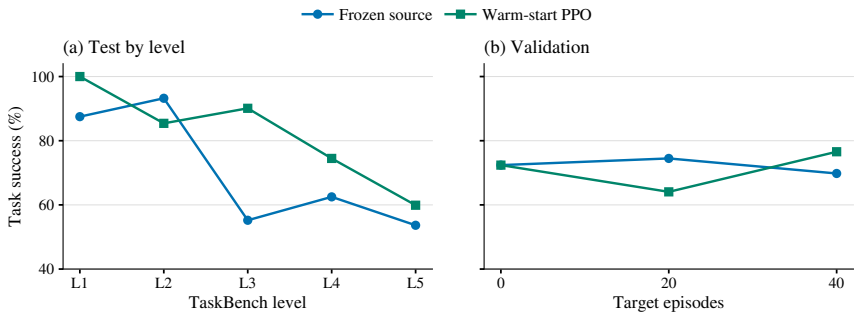


Figure 7: Cross-dataset transfer to TaskBench. (a) Frozen Overcooked C2 policy and the same policy after 40 target training episodes, by target level. (b) Validation at 0, 20, and 40 training episodes on eight separate combinations.

the C5 policy obtains 54% on C1 compared with 98% for the C1 specialist. In these runs, training on the hardest scenario alone does not recover performance on easier ones.

5.6 Cross-Dataset Transfer and Adaptation

Because all domains share the allocation interface, a policy trained in Overcooked can be deployed on other datasets without modification. TaskBench is our most detailed extension and is used for both frozen transfer and adaptation; Minecraft and Factorio serve as representative feasibility checks (Appendix G).

A frozen source policy remains operational across the evaluated domains. Without target training, the Overcooked C2 policy achieves 70% success on native TaskBench tool workflows, 88/93% on L1/L2 and 54–63% on L3–L5 (Figure 7a). The same policy reaches 90% success on Minecraft and 81% on Factorio.

Limited target adaptation improves average TaskBench success. Warm-starting with 40 target episodes (four PPO updates) raises TaskBench success from 70% to 82%. Changes vary across levels: L3 gains 35 points, while L2 declines by 8 points, and validation success dips after 20 episodes before exceeding its initial value after 40 (Figure 7b).

6 Conclusion and Limitations

Schedule-Overcooked isolates multi-task allocation by fixing decomposition and execution, grades difficulty through composable agent, task, and resource constraints, and exposes a shared allocation interface across domains. In our experiments, high agent utilization often coexists with unfinished workflows, the evaluated allocators vary substantially across constraint scenarios, a learned allocator shows low success already on its training configurations once temporal constraints are combined with resource competition, and frozen policies transfer well across workload levels but asymmetrically across constraint scenarios. These results suggest several directions. Identifying why learned allocators stall under compound constraints, for example by separating temporal credit assignment from exploration and representation effects, remains open. Revealing capability information does not help the tested policies at evaluation time; whether policies trained to use such information benefit from it is untested. Adaptation to TaskBench improves average success but shifts performance across levels, so retaining source behavior while adapting is an open question. Finally, extending the Minecraft and Factorio evaluations to broader native workloads is a natural next step.

7 Ethics Statement

This work studies task allocation in simulated kitchen environments and tool-using multi-agent workflows. The benchmark is intended for research on orchestration efficiency and does not establish the safety of deploying autonomous agents in real-world settings. In practical applications, allocation objectives should account for fairness, privacy, and safety alongside efficiency. Integration with external tools should retain appropriate access controls and human oversight, particularly when agent actions can affect people or shared resources.

8 Reproducibility Statement

The benchmark code, task configurations, maps, and baseline implementations are available at `https://anonymous.4open.science/r/Schedule-Overcooked-2BE9/`. Appendix B defines task generation, timing, and execution contracts; Appendix C describes the shared runtime; Appendix E gives the policy architecture and optimization settings. Appendices F and G specify evaluation cohorts, checkpoint selection, metrics, and transfer configurations. The supplementary archive contains the manuscript sources and the numerical summaries used for the reported figures and tables.

9 AI Use Statement

We used large language models to polish the writing of this paper and to assist in generating data-visualization code. The authors reviewed all AI-assisted content and take full responsibility for the paper and its results.

References

A2A Project. Agent2Agent (A2A) protocol specification, version 0.3.0. Protocol documentation, 2026. URL `https://a2a-protocol.org/v0.3.0/specification/`. Accessed September 13, 2026.

Muzhen Cai, Xiubo Chen, Yining An, Jiaxin Zhang, Xuesong Wang, Wang Xu, Weinan Zhang, and Ting Liu. CookBench: A long-horizon embodied planning benchmark for complex cooking scenarios. *arXiv preprint arXiv:2508.03232*, 2025. URL `https://arxiv.org/abs/2508.03232`.

Shijie Cao, Yuan Yuan, and Jing Liu. DynaSchedBench: Calibrated dynamic scheduling benchmarks and observability paradox in LLM-based scheduling agents. *arXiv preprint arXiv:2605.27566*, 2026. URL `https://arxiv.org/abs/2605.27566`.

Micah Carroll, Rohin Shah, Mark K. Ho, Thomas L. Griffiths, Sanjit A. Seshia, Pieter Abbeel, and Anca Dragan. On the utility of learning about humans for human-AI coordination. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL `https://arxiv.org/abs/1910.05789`.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023. URL `https://arxiv.org/abs/2305.05176`.

Yubo Dong, Xukun Zhu, Zhengzhe Pan, Linchao Zhu, and Yi Yang. VillagerAgent: A graph-based multi-agent framework for coordinating complex task dependencies in Minecraft. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 16290–16314. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.964. URL `https://aclanthology.org/2024.findings-acl.964/`.

Gonzalo Gonzalez-Pumariega, Leong Su Yean, Neha Sunkara, and Sanjiban Choudhury. Robotouille: An asynchronous planning benchmark for LLM agents. In *International Conference on Learning Representations*, 2025. URL `https://arxiv.org/abs/2502.05227`.

Jack Hopkins, Mart Bakler, and Akbir Khan. Factorio learning environment. *arXiv preprint arXiv:2503.09617*, 2025. URL https://arxiv.org/abs/2503.09617.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. *arXiv preprint arXiv:2408.08435*, 2024. URL https://arxiv.org/abs/2408.08435.

Oucheng Huang, Yuhang Ma, Zeng Zhao, Mingrui Wu, Jiayi Ji, Rongsheng Zhang, Zhipeng Hu, Xiaoshuai Sun, and Rongrong Ji. ComfyGPT: A Self-Optimizing Multi-Agent System for Comprehensive ComfyUI Workflow Generation. *arXiv preprint arXiv:2503.17671*, 2025. URL https://arxiv.org/abs/2503.17671.

Dingkang Liang, Cheng Zhang, Xiaopeng Xu, Jianzhong Ju, Zhenbo Luo, and Xiang Bai. Cook and clean together: Teaching embodied agents for parallel task execution. *arXiv preprint arXiv:2511.19430*, 2025. URL https://arxiv.org/abs/2511.19430.

Wenrui Liao, Weihong Du, Yi Li, Hongru Liang, and Wenqiang Lei. NL → Schedule: Evaluate multitask scheduling capability of large language models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 35620–35640, 2026. doi: 10.18653/v1/2026.acl-long.1648. URL https://aclanthology.org/2026.acl-long.1648/.

Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony G. Cohn, and Janet B. Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 30108–30134. PMLR, 2024. URL https://proceedings.mlr.press/v235/lin24k.html.

Jiayu Liu, Cheng Qian, Zhaochen Su, Qing Zong, Shijue Huang, Bingxiang He, and Yi R. Fung. CostBench: Evaluating multi-turn cost-optimal planning and adaptation in dynamic environments for LLM tool-use agents. *arXiv preprint arXiv:2511.02734*, 2025. URL https://arxiv.org/abs/2511.02734.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general ai assistants. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/25ae35b5b1738d80f1f03a8713e405ec-Abstract-Conference.html.

Fan Nie, Lan Feng, Haotian Ye, Weixin Liang, Pan Lu, Huaxiu Yao, Alexandre Alahi, and James Zou. Weak-for-Strong: Training Weak Meta-Agent to Harness Strong Executors. *arXiv preprint arXiv:2504.04785*, 2025. URL https://arxiv.org/abs/2504.04785.

Shuofei Qiao, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Benchmarking agentic workflow generation. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.07869.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL https://arxiv.org/abs/1707.06347.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. TaskBench: Benchmarking large language models for task automation. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/085185ea97db31ae6dcac7497616fd3e-Abstract-Datasets_and_Benchmarks_Track.html.

Xi Shi, Mengxin Zheng, and Qian Lou. Learning latency-aware orchestration for multi-agent systems. *arXiv preprint arXiv:2601.10560*, 2026. URL https://arxiv.org/abs/2601.10560.

Wen Song, Xinyang Chen, Qiqiang Li, and Zhiguang Cao. Flexible job-shop scheduling via graph neural network and deep reinforcement learning. *IEEE Transactions on Industrial Informatics*, 19 (2):1600–1610, 2023. doi: 10.1109/TII.2022.3189725. URL https://ieeexplore.ieee.org/document/9826438/.

Haochen Sun, Shuwen Zhang, Lujie Niu, Lei Ren, Hao Xu, Hao Fu, Fangkun Zhao, Caixia Yuan, and Xiaojie Wang. Collab-overcooked: Benchmarking and evaluating large language models as collaborative agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 4922–4951, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.249. URL https://aclanthology.org/2025.emnlp-main.249/.

Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002. doi: 10.1109/71.993206. URL https://ieeexplore.ieee.org/document/993206/.

Kun Wang, Guibin Zhang, ManKit Ye, Xinyu Deng, Dongxia Wang, Xiaobin Hu, Jinyang Guo, Yang Liu, and Yufei Guo. MAS^2: Self-Generative, Self-Configuring, Self-Rectifying Multi-Agent Systems. *arXiv preprint arXiv:2509.24323*, 2025a. URL https://arxiv.org/abs/2509.24323.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. ScoreFlow: Mastering LLM Agent Workflows via Score-based Preference Optimization. *arXiv preprint arXiv:2502.04306*, 2025b. URL https://arxiv.org/abs/2502.04306.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. MAS-GPT: Training LLMs to Build LLM-based Multi-Agent Systems. *arXiv preprint arXiv:2503.03686*, 2025. URL https://arxiv.org/abs/2503.03686.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B. Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models. In *International Conference on Learning Representations*, 2024a. URL https://arxiv.org/abs/2307.02485.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. *arXiv preprint arXiv:2410.10762*, 2024b. URL https://arxiv.org/abs/2410.10762.

Shiqi Zhang, Xinbei Ma, Yunqing Xu, Zouying Cao, Pengrui Lu, Haobo Yuan, Tiancheng Shen, Zhuosheng Zhang, Hai Zhao, and Ming-Hsuan Yang. ParaCook: On time-efficient planning for multi-agent systems. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 27683–27701, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1378. URL https://aclanthology.org/2026.findings-acl.1378/.

Yikai Zhang, Siyu Yuan, Caiyu Hu, Kyle Richardson, Yanghua Xiao, and Jiangjie Chen. TimeArena: Shaping efficient multitasking language agents in a time-aware simulation. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3894–3916, Bangkok, Thailand, August 2024c. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.215. URL https://aclanthology.org/2024.acl-long.215/.

Mingming Zhao, Xiaokang Wei, Yuanqi Shao, Kaiwen Zhou, Lin Yang, Siwei Rao, Junhui Zhan, and Zhitang Chen. A^2Flow : Automating Agentic Workflow Generation via Self-Adaptive Abstraction Operators. *arXiv preprint arXiv:2511.20693*, 2025. URL https://arxiv.org/abs/2511.20693.

Heng Zhou, Hejia Geng, Xiangyuan Xue, Li Kang, Yiran Qin, Zhiyong Wang, Zhenfei Yin, and Lei Bai. ReSo: A reward-driven self-organizing LLM-based multi-agent system for reasoning tasks. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 15979–15998, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.808. URL https://aclanthology.org/2025.emnlp-main.808/.

A Evidence for the Empirical Observations

This appendix provides the configurations and results underlying the observations in Section 3.1. These diagnostic cohorts are separate from the suite evaluations in Section 5.

A.1 Benchmark Coverage of Concurrent Agent Allocation

Comparison criteria. Table 1 uses the following definitions. **Multi-task** requires distinct goals sharing an execution timeline. **Agent team** requires scheduling a team of identified execution agents, rather than assigning work to abstract compute nodes or machines. **Decoupled evaluation** assesses assignments under fixed execution, including machine assignments where applicable. **Transfer evaluation** requires an explicit generalization evaluation beyond the source task distribution, such as held-out task families or domains. **Levels** requires explicit difficulty strata or complexity-indexed evaluation. **Multi-constraint** requires multiple interacting constraint types, such as dependencies, occupancy, and resource availability. A check marks a documented protocol meeting the criterion; a cross marks its absence. Our row covers the framework and the cross-domain tests in Section 5.6.

GAIA and SciBench-MAS. GAIA evaluates individual questions at three difficulty levels, with no agent-team scheduling interface (Mialon et al., 2024, Section 3.3). SciBench-MAS evaluates agent selection within a composite reasoning task at easy, medium, and hard levels. ReSo additionally evaluates standard benchmarks in different domains, while SciBench-MAS itself specifies within-benchmark difficulty splits (Zhou et al., 2025, Sections 5.1–5.4).

ParaCook and TimeArena. ParaCook reports easy, medium, and hard cooking settings. Its evaluated agent-indexed action plans couple allocation with low-level execution (Zhang et al., 2026, Sections 4.4, 6, and 7). TimeArena grades workload by single, double, and triple tasks, with occupancy, duration, and shared-object constraints for one agent (Zhang et al., 2024c, Sections 3–5). Neither protocol specifies source-to-target task transfer.

CostBench and DynaSchedBench. CostBench evaluates complexity through task-sequence length and combines tool prerequisites, costs, availability, and dynamic blocking (Liu et al., 2025, Sections 3 and 5). DynaSchedBench grades scheduling difficulty through its Schedule Stress Index and dispatches jobs to machines in a fixed simulator (Cao et al., 2026, Sections 3–4). Both evaluate variation within their respective domains, without a specified task-family transfer protocol.

ORS3D-60K and CookBench. ORS3D-60K grades composite goals by subtask count and reports separate results for each count (Liang et al., 2025, Sections 3.4 and 5.4). Its scheduling model controls foreground/background occupancy, with separate object-localization scoring. CookBench grades recipes by preparation difficulty and instructions by semantic difficulty (Cai et al., 2025, task-framework section). Its embodied track adds ingredient, spatial, and cooking constraints. Neither study specifies a held-out task-family transfer evaluation.

AsyncHow. AsyncHow evaluates workflow-duration reasoning across graph-complexity levels. Its out-of-distribution probe varies graph complexity within the same temporal-reasoning formulation; resources are assumed unlimited (Lin et al., 2024, Sections 4.2 and 5.3).

WorFBench. WorFBench evaluates workflow generation and tests trained models on held-out task datasets, including Seal-Tools and InterCodeSQL (Qiao et al., 2025, Sections 2.2 and 4.1). The output specifies workflow structure rather than execution-agent assignments or interacting operational constraints.

Robotouille. Robotouille provides increasingly complex recipe families and paired synchronous/asynchronous variants, without a common difficulty-level protocol. Its multi-agent track supports agent-indexed actions, occupancy, and asynchronous processing (Gonzalez-Pumariega et al., 2025, Sections 2–3 and Appendices A.3 and A.6.3). The primary evaluation uses a single agent and specifies no task-transfer split.

A.2 ParaCook Allocation and Execution Diagnostic

This diagnostic compares one-shot embodied action plans with online subtask–agent assignments executed by a shared programmatic adapter. The adapter performs navigation, interaction, processing, and synchronization. The 45 test cases contain 16 Easy, 15 Medium, and 14 Hard configurations, with two to four agents and two to five orders.

Table 5: Reported complete-episode successes in the ParaCook diagnostic.

Allocator identifier	Action plans	Online allocation
GPT-5.4	4/45	30/45
Qwen3.7-Plus	7/45	26/45
Gemini-3.1-Pro-Preview	31/45	45/45

These counts support Observation 1's finding that outcomes depend on the execution interface. Remaining failures in the online condition include dependency violations and blocked storage, plates, or equipment, supporting Observation 2. This diagnostic is separate from the suite comparisons in Section 5.

A.3 Feasibility Support in Learned Allocation

Setting. This experiment uses the decoupled ParaCook setting of Appendix A.2: allocators select subtask–agent pairs online, and the shared programmatic adapter executes them. Learned allocators train on 130 cases and are evaluated on the same 45 test cases (16 Easy, 15 Medium, 14 Hard) as the LLM allocators. A case succeeds when all of its orders are completed.

Feasibility masks. Each allocator scores only the subtask–agent pairs admitted by a mask. The **weak mask** retains DAG precedence: a subtask is admissible once its predecessors are complete. The **strong mask** additionally checks agent state and physical storage and equipment conditions.

Allocators. We compare an **MLP** pair-scoring policy and a **GraphPlanner-style** policy, whose "Graph" component retrieves historical scheduling memory; a **NoGraph** variant removes this memory. Policies are trained by behavioral cloning (BC), by PPO from scratch, or by PPO initialized with a BC policy, optionally with a critic warm-up. A CP-SAT solver and **OMAC-style** prompt refinement of an LLM allocator serve as references.

Findings. Table 6 supports Observation 3 in three ways. First, **feasibility support is decisive**: the strong mask raises MLP success from 14 to 37 cases, and a BC policy trained with the strong mask drops from 34 to 12 cases when evaluated under the weak mask. Without state-aware masking, learned policies can select pairs that cannot start. Second, **initialization matters for RL**: PPO from scratch succeeds on 14 cases, BC initialization raises this to 32, and adding a critic warm-up reaches 41, exceeding the BC policy itself. Third, the lightweight learned allocator (41/45) exceeds GPT-5.4 (30/45), Qwen3.7-Plus (26/45), and prompt refinement (23/45), approaching Gemini-3.1-Pro-Preview and CP-SAT (45/45) in this compact setting; removing the retrieved scheduling memory leaves its success unchanged. These results show that trained lightweight allocators can handle settings with few constraints when feasibility is modeled explicitly, which motivates the constraint-graded study of learned allocation in Section 5.3.

Table 6: Successful cases on the 45-case decoupled ParaCook test set.

Allocator	Mask (training / evaluation)	Success
CP-SAT	–	45/45
OMAC-style prompt refinement	–	23/45
MLP	strong	37/45
MLP	weak	14/45
GraphPlanner BC	strong / strong	34/45
GraphPlanner BC	weak / weak	19/45
GraphPlanner BC	strong / weak	12/45
GraphPlanner PPO, no BC	strong	14/45
GraphPlanner PPO, policy BC	strong	32/45
GraphPlanner PPO, policy BC + critic warm-up	strong	41/45
NoGraph PPO, policy BC + critic warm-up	strong	41/45
PPO, policy BC + critic warm-up, 1,000 episodes	weak	4/45

A.4 TaskBench Temporal Allocation Diagnostic

The diagnostic fixes 17 TaskBench workflows and their 47 tool nodes, with original dependencies preserved (Shen et al., 2024). Six workers have overlapping tool permissions and one foreground slot each. Video, decoding, audio, and image services each have capacity one; downloads share two sessions and 2 MiB/s bandwidth. Tasks arrive at 0, 8, and 16 seconds, one worker is unavailable during seconds 12–28, and download admission pauses during seconds 8–18.

Workers use Qwen3.8-Flash and invoke native FFmpeg or local HTTP tools. Each subtask has an eight-second response window followed by a 1–8-second service floor; overruns retain actual duration. The horizon is 600 seconds. Deadlines are fixed before evaluation, and expired tasks remain active. All 187 recorded worker calls pass output and execution checks. Serial completes one high-level task before moving to the next, allowing within-task parallelism; this is distinct from the strict single-subtask Serial baseline in Section 5.1.

Table 7: Outcomes for the 17-task diagnostic.

Allocator	Completed	On time	Elapsed time (s)
Serial orders	15/17	1/17	600
Random	17/17	15/17	195.50
Greedy-Cost	17/17	15/17	166.98
EDF	17/17	14/17	191.11

Greedy-Cost and EDF complete all workflows but miss different deadlines, supporting Observation 2. All rules use the same fixtures and worker setting.

B Benchmark Configuration and Execution Contracts

B.1 Basic Task Suite

The basic suite has ten core tasks per B1–B4 level. B1 uses compact single-chain recipes; B2 increases concurrent task count on paired layouts; B3 adds longer chains and multi-branch joins; B4 preserves the B3 task DAGs and functional resource quantities while enlarging layouts and adding scene descriptions. Three terrain variants and five tool profiles produce 150 configurations per level. The official core flag identifies the prescribed terrain/profile for each template.

B.2 Five-Level Constraint Suite

The constraint suite has five core families per level. C1 uses basic skill matching and finite capacity; C2 increases resource competition; C3 adds arrivals and task deadlines; C4 adds local expiry and outages; C5 adds hidden success rates and deadline-induced fires. C2–C4 use paired layouts, resources, and agent configurations. C5 retains its own layouts and arrivals.

For C3–C5, task deadlines are 192 steps after arrival. C4/C5 additionally use a 60-step acknowledgment deadline and a 30-step cooked-output collection window. The latter starts after processing completes. Episodes can finish when all tasks end; otherwise their horizons are the last task deadline plus one step.

Table 8: Constraint-suite horizons, in simulation steps.

Level	M1	M2	M3	M4	M5
C1	450	450	450	450	450
C2	500	500	550	700	550
C3	293	343	343	343	393
C4	293	343	343	343	393
C5	357	377	395	383	365

B.3 Subtask Contracts

Each subtask specifies an operation, arguments, input/output identities, predecessors, capability requirements, and a completion verifier. Source-to-runtime expansion preserves all-input barriers. Transfers are created when selected ownership makes an input inaccessible.

Table 9: Source operation contracts.

Operation	Required outcome
`get_raw_ingredient`	Fetch the specified ingredient.
`get_empty_dish`	Obtain a clean dish artifact.
`cook_and_delivery`	Process specified inputs with a compatible tool.
`serve`	Plate or submit the specified dish.
`serve_customer`	Complete the task's service operation.
`checkout`	Complete the designated checkout.
`wash_dish`	Restore a clean plate from dirty stock.
`fire_handle`	Extinguish the specified fire.
`trash_handle`	Remove designated waste.

Completion requires the matching physical transition and artifact identity. Background processing can release an agent while retaining equipment. Produced artifacts retain their storage reservation until pickup. Dependencies, resource capacities, and input ownership are validated before execution; outcome checks determine delivery and deadline compliance.

C Allocation Support Framework and A2A Integration

C.1 Responsibilities and Decision Interface

We consistently distinguish **task decomposition**, which supplies task DAGs, **task allocation**, which selects subtask owners and dispatch times, and **task execution**, in which agents perform their assigned work. Orchestration denotes the overall coordinating framework. The fixed executor resolves local navigation, resource binding, and tool invocation for the selected agent. Allocation is evaluated under this common execution policy, including its time costs and failure conditions.

A decision carries the current observation revision and an ordered list of subtask–agent pairs. The following schematic dispatch message binds one known node to one known agent:

```
{"observation_id": 12,
 "dispatch": [{"task_id": "O0.process", "agent_id": "A1"}]}
```

Identifiers must exist in the current observation. The operation and its arguments are specified by the node contract. The revision detects stale decisions. LLM, rule-based, and learned allocators share this boundary even when their internal state representations differ.

C.2 Allocation Trigger

The trigger compares a decision-state signature containing released task states, artifact locations and readiness, reservations, workstation availability, and agent availability. Table 10 lists the invo-

cation conditions. The event-only protocol combines state changes with initialization and rejection recovery.

Table 10: Allocator invocation conditions. State-change rows refer to changes in the decision-state signature; invocation remains subject to episode and decision budgets.

Event	Invocation condition
Initialization	First allocation in the episode.
Task release	A task enters the released task set.
Task transition	Completion, failure, cancellation, or a changed binding or blocking state.
Agent availability	Execution capacity, held artifact, or recovery state changes.
Artifact or resource change	Input readiness, artifact location, reservation, or workstation availability changes.
Dispatch continuation	An accepted nonempty batch permits a follow-up allocation at the same simulation time.
Rejected dispatch	A rejected batch schedules a retry after one simulation step.

After a dispatch batch, the comparison signature is updated to the resulting state. Ordinary movement and logging updates are filtered by this signature. The current loop permits up to 32 consecutive decisions at one simulation time, ending the sequence when a batch is empty or rejected. Adapters without a decision-state signature use execution events and task releases as the fallback trigger.

In a synchronous evaluation, environment advancement and allocation follow the configured simulation clock. A real-time backend instead continues execution while the allocator computes. Its integration must define how stale revisions are handled and measure decision latency separately from simulated operation duration. Under partial observation, the trigger uses only changes permitted by the observation protocol.

C.3 DAG Manager: Validation and Deferred Execution

The runtime checks identity, revision, release time, predecessor completion, skills, capacity, input access, and resource conflicts. It tracks dependencies, artifact ownership, and resource reservations to maintain ready subtasks and eligible subtask–agent pairs as execution advances. Validation is transactional: bindings are committed only when the entire batch passes. Rejections return a reason and relevant blocking information. Candidate masking precedes final batch validation. Accepted bindings wait for any required input transfers before execution.

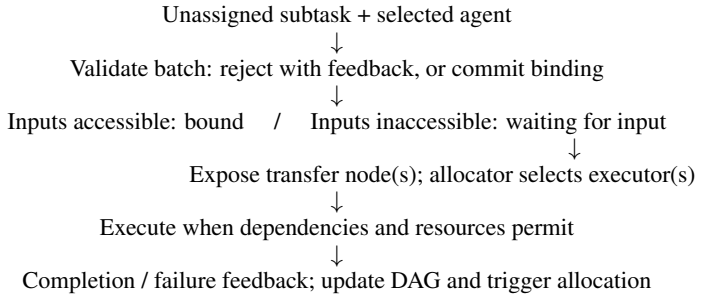


Figure 8: Allocation and execution lifecycle. Waiting retains the consumer's selected owner while transfer ownership remains an explicit allocation decision. Background operations can release agent capacity before completion.

Suppose node u, executed by A_0, produces artifact x, and its consumer v is subsequently allocated to A_1. If A_1 cannot access x, the runtime creates a transfer node h, augments the dependency chain to $u \to h \to v$, and keeps v waiting with owner A_1. The allocator then selects an eligible agent for h. Completing h makes the artifact accessible; v resumes when the remaining execution conditions hold, without requiring a new owner decision. Transfer routes are bounded (the generic runtime defaults to eight transfer segments per consumer binding); unreachable or excessive routes are rejected.

Pending bindings, foreground work, and background work have distinct capacity semantics. Equipment may remain occupied after an agent is released. A bounded recovery mechanism can reopen safely releasable unexecuted bindings in a closed wait cycle while preserving completed transfers

and real artifacts. Such state changes remain visible in feedback, with subsequent ownership decisions made by the allocator.

C.4 Assignment-to-Action Adapter

The adapter consumes a fixed subtask contract and the selected owner. It resolves local paths, station interactions, input pickup, processing, and delivery according to a common execution policy. It preserves artifact provenance across processing and plating so that another order's leftover product cannot satisfy the selected task. Completion requires the appropriate physical transition and output identity.

The allocator selects business subtasks, owners, dispatch times, and batch order; the adapter selects local paths and compatible facilities. Execution starts after required input transfers complete and resources become available. Movement, waiting, resource occupancy, and faults retain their physical costs.

Configured reliability is sampled at the first actual business attempt, before its side effects. A successful admission authorizes that binding; internal navigation, repeated observation, and resumption of background work do not independently resample it. Explicit transfer tasks have their own contracts and attempts. Real outages, expiry, resource blocking, and other configured environment events still apply after admission. The basic B workload uses deterministic capability execution; the C suite retains its configured operation-success probabilities.

An accepted assignment may wait, expire, or fail under these rules. The shared adapter fixes the execution policy across methods. Replacing the fixed adapter with LLM workers defines a separate end-to-end executor condition, reported separately from allocation-only comparisons.

C.5 Fixed Local Decisions

Table 11: Ownership of allocation and local execution decisions. All methods share the same versioned rules within a comparison.

Object	Allocator	Shared framework
Business progress	Subtask, owner, dispatch time, and batch order	Validate contracts and batch conflicts; preserve task and owner.
Navigation	Selected operation and executor	Local path planning, avoidance, and interactions; charge physical time.
Facilities	Resource demand induced by chosen work	Bind compatible, currently available facilities by fixed local rules.
Handoffs	Explicit transfer executor and dispatch	Generate required transfer nodes and a feasible connection after ownership is selected.
Recovery	Next allocation after feedback	Release safely reversible bindings under a fixed rule; report changes.

For processing facilities, the source adapter filters compatibility, reachability, availability, and reservations, then orders candidates by distance and stable identifier. Dish sources use the same distance/identifier ordering. Output storage prefers an input counter that can be reclaimed. Otherwise it orders empty buffers by the number of agents that can reach them, distance, and identifier, favoring less-shared storage first. Binding within a batch follows the submitted order and shared validation rules. These mechanisms are part of the fixed execution policy shared by all methods.

Local rules preserve the selected business operation and task priorities while choosing exact facilities. Section 5.4 examines task and resource restrictions through paired constraint interventions.

C.6 A2A Mapping and Portability

Our tool adapter uses the official A2A Python SDK with the A2A 0.3.0 JSON-RPC interface (A2A Project, 2026). Agent Cards describe skills and endpoints. A selected node becomes a `Message` carrying a structured `DataPart`; the worker returns an A2A `Task`, its status, and output `Artifacts`. A schematic application payload is:

```
{"node_id": "O0.process", "attempt": 0,
 "operation": "process", "arguments": {"inputs": ["artifact-x"]}}
```

The payload is application data carried inside a JSON-RPC message. Workflow-node identity and execution-attempt identity are tracked separately. Dependencies, deadlines, resource occupancy, and empirical capability estimates are application metadata. Input-wait and background states are tracked internally alongside standard A2A task states.

An integration adapter exposes these allocation facts and execution events; workers retain their existing local tools, including MCP-based interfaces. With shared artifact access, no transfer is needed. With private workspaces, a transfer can represent an explicit copy or import. The TaskBench implementation integrates A2A JSON-RPC over in-process HTTP ASGI transport.

D Extension Task and Execution Specifications

The extensions preserve the allocation boundary of Section 4.4: policies select subtask–worker pairs and dispatch timing, while native adapters perform operations and verify outcomes.

D.1 Shared Contract

Each task has a fixed operation DAG with typed inputs and outputs. Workers have separate identities, capabilities, and foreground capacity. Admission checks dependencies, input ownership, and joint resource requirements. Receipts record before/after state, attempt identity, timestamps, resource use, and completion verification. Failed admission, task failure, and infrastructure faults are recorded separately. Initialization resets task-specific inventories and outputs.

D.2 TaskBench

TaskBench supplies tool-graph workflows (Shen et al., 2024). The evaluated extension executes media and download operations using native FFmpeg and a local HTTP service. Subtasks specify tool arguments and artifact identities; services have concurrency limits, and workers have tool permissions. Typed output validation determines subtask completion. Task clocks include native service timing. The adaptation experiment uses the resource-pool graph adapter and source-style sampled decoding. Its paired training and evaluation protocol is given in Appendix F.

D.3 Minecraft

The VillagerBench-based extension (Dong et al., 2024) uses native Minecraft Java execution. Contracts cover material retrieval, transport, crafting/processing, and block placement. Retrieval and transport reconcile container and worker inventories; crafting consumes recipe inputs and verifies outputs; placement checks target coordinates and material use. The representative transfer study uses farm and factory task families at L2/L5, with six orders per case. Resource limits, inventories, worker skills, outages, collection windows, and task deadlines are supplied in each configuration. Native reference runs precede evaluation. Results and concrete capacities are in Appendix G.

D.4 Factorio

The Factorio extension uses native production and inventory transitions, following the setting of Hopkins et al. (2025). Tasks cover material supply, smelting, component production, assembly, transport, and final delivery. Public operation graphs are exposed after task acceptance. Completion receipts verify the delivered item type, quantity, and order identity. Machine compatibility, storage, worker inventory, processing time, and outages constrain admission and execution. The fixed adapter performs joint compatible-equipment binding after the policy selects task–worker pairs. The evaluated graph uses capacity-unit resource nodes; weights and the source decoder remain fixed. The compact transfer cohort is described in Appendix G.

E GNN-PPO Architecture and Training

E.1 Graph and Action Representation

The baseline adapts the heterogeneous graph network of Song et al. (2023). Task, agent, resource, and artifact nodes use 48-dimensional input features. Relations encode precedence, eligibility, resource use, production, consumption, and access. Two message-passing layers have width 64. The actor scores subtask–agent pairs from node embeddings, pooled task/agent context, batch-prefix state, timing features, and capability statistics. A separate head scores `STOP`. Prefix masks enforce compatibility within an autoregressive dispatch batch. In the reported event-only protocol, the learned WAIT head is disabled; the shared trigger determines reinvocation.

E.2 Capability Statistics

Episode-local public attempt feedback updates $p_{a,o} \sim \mathrm{Beta}(1 + s_{a,o}, 1 + f_{a,o})$, where $s_{a,o}$ and $f_{a,o}$ count successful and failed attempts by agent a on operation o. The actor receives the posterior mean $\mu_{a,o}$ and standard deviation $\sigma_{a,o}$. Visible rates use their disclosed value with zero uncertainty. The full capability head adds

$$g_\mu(z) \log \max(\mu_{a,o}, 10^{-6}) + g_\sigma(z)\sigma_{a,o}$$

to a pair logit, where both gates are $2\,\mathrm{sigmoid}$ transformations of the pair context. Feedback is public; private success rates are hidden on C5.

E.3 Objective and Optimization

The reported source runs optimize task completion with a small flow penalty:

$$R = \frac{N_{\mathrm{complete}}}{N} - 0.01F, \qquad F = \frac{1}{NH}\sum_i \max(0, \tau_i - r_i).$$

For a completed task, $\tau_i = \min(C_i, H)$; otherwise $\tau_i = H$. Thus unfinished work is charged through the horizon even if execution terminates early. These runs use zero auxiliary progress and counter-release reward. Returns-to-go include only rewards after the recorded decision revision, preventing credit for outcomes already visible in the observation.

Training uses clipped PPO (Schulman et al., 2017) with a learned critic, undiscounted Monte Carlo targets, equal episode loss weights, and standardized advantages. The joint likelihood includes all selected pairs and STOP. All encoder, actor, capability-head, and critic parameters are trainable.

Table 12: Source PPO training settings.

Setting	Value
Episodes per update	20
Maximum updates	200
Validation interval	20 updates
PPO passes / minibatch size	2 / 32
Adam learning rate	2×10^{-4}
Clipping / entropy coefficient	0.2 / 0.005
Value-loss / behavior-KL coefficient	0.5 / 0.1
Gradient norm bound	1

Checkpoints maximize source validation completion, retaining the earlier checkpoint on a tie. Plateau stopping begins after update 100 with patience five validation checks and improvement threshold 0.005. The common plotted window ends at update 180. Target adaptation instead uses four ten-episode updates at learning rate 10^{-4}, as specified in Appendix F.

F Evaluation Protocols and Detailed Results

This appendix specifies the evaluation cohorts, aggregation rules, and checkpoint selection used in the main text. Numerical summaries are included in the anonymous supplementary archive.

Basic-suite comparison. The rule-based comparison contains 200 method–case pairs (40 cores and five policies). The GNN-PPO row uses the B1–B4 specialists selected at updates 160, 200, 60, and 200, evaluated on official core configurations. Core membership follows the dataset's explicit flag, including each task's prescribed terrain and tool profile. Per-case SR and occupied agent-time are included in the supplementary data.

Agent utilization. We use ParaCook's per-agent time normalization (Zhang et al., 2026, Equation 8), computed over all valid episodes. For an episode of duration T with M agents, $\mathrm{AU} = (MT)^{-1} \sum_{a=1}^{M} \int_0^T b_a(t)\, dt$, where $b_a(t)$ indicates foreground occupancy. Occupancy includes navigation, interaction, and blocking while an agent retains a running assignment; background processing after agent release is excluded. We integrate recorded lifecycle events and check final assignment and occupancy states against the saved observation. Per-case rates use the same episodes as task success, followed by equal weighting across cases.

Learning curves. Basic models train separately on B1–B4 under the basic-suite deadline setting. Constraint models train separately on C1–C5 on the v3.6 dataset. Each basic task family supplies ten training, two validation, and three test variants. Constraint families use separate terrain variants for training, validation, and testing. These splits retain shared task families and evaluate configuration generalization. All runs use the settings in Table 12, with validation every 20 updates. Test checkpoints are selected by source validation performance; target scores never enter source checkpoint selection. Figure 4 plots all histories through the common update 180. Faint traces show raw training means and solid curves trailing five-update means; validation is unsmoothed.

Constraint baselines. Table 4 uses the v3.6 core campaigns for Serial, Random-Feasible, EDF, Greedy-Time, and a resource-aware HEFT adapter, each with five maps per level. GNN-PPO cells use validation-selected checkpoints tested on five maps. LLM allocators (Gemini, Grok, and GPT-Sol) share the same timing specification. Both SR and AU use the same episodes. The main tables display integer percentages; highlighting uses the unrounded values.

LLM allocators. Gemini 3.8 Flash, Grok 4.6, and GPT-5.6-Sol are queried through an OpenAI-compatible chat-completion interface with temperature 0, a JSON-object response format, a budget of 8,192 completion tokens, and each provider's default reasoning setting. Each call contains a fixed system prompt and the current public observation serialized as JSON. The system prompt states the objective (maximize orders finished before their deadlines, then reduce completion time), the batching and transfer rules, the meaning of running and background occupancy, and that an empty dispatch means waiting. The observation lists tasks with their state, required skill, dependencies, deadline, and input/output artifacts; agents with capability tags and, where observable, estimated success rates; artifacts, reservations, and resource states (availability, occupancy, storage, outages, and fires); execution-cost hints; the admissible subtask–agent pairs; and the eight most recent rejections and orchestration events. The model returns {`observation_id, dispatch: [{task_id, agent_id}], reconsider_after`}, where the optional `reconsider_after` requests a later reinvocation. LLM allocators use the same trigger and validator as all other methods and are not called when no admissible pair exists. A rejected batch returns its reason in the next observation. Transport errors and unparseable responses are retried up to eight times with exponential backoff, doubling the token budget (up to 32,768) after malformed output.

HEFT behavior. HEFT ranks all unassigned, schedulable subtasks of all released orders jointly by upward rank, using the public cost estimates, and recomputes this plan at every invocation; each call dispatches the planned pair with the earliest finish time. Because ranks follow critical-path length over the union of orders, the plan interleaves subtasks from every order. In a B3 case with six orders and two agents, for example, HEFT completes 24 subtasks spread across all six orders but finishes none of them, whereas EDF completes two orders. Its agent calendars also do not model how long produced artifacts occupy storage, so intermediate products can fill all buffers and block downstream plating. Agents therefore remain occupied, yielding near-full utilization, while few workflows complete before their deadlines.

Controlled interventions. Figure 5 uses checkpoints selected by source validation, at updates 120–200. Each intervention pairs the original and modified configuration on all five families while

keeping checkpoint weights and horizon fixed; the mean deadline effects are 20.67, 17.13, and 13.56 percentage points for C3–C5. The intervention cohort's original SR is 29.71%/33.92%/16.51%, and removing task deadlines gives 50.38%/51.04%/30.07%. Full checkpoint responses appear in Tables 13–15.

Cross-level transfer. Figure 6 uses paired official core cases: ten B cores per level and five C families per level. The B matrix includes four specialists and one policy trained across B1–B4. Weights are fixed and selected on source validation. Figure 9 also reports B-to-C and C-to-B transfer.

TaskBench transfer and adaptation. Figure 7 compares a frozen source policy with the same policy after target adaptation. The source checkpoint is C2 at update 120, selected only on source validation. All policies retain source-style categorical sampling and the resource-pool graph adapter. Warm-starting uses 40 mixed-task target episodes and four PPO updates with the unchanged architecture and learning rate 10^{-4}; the final checkpoint is fixed at update four. Training, validation, and final testing use composition variants 0, 1, and 2. The final test contains 20 combinations. Validation contains eight L2/L4 combinations and is recorded at 0, 20, and 40 training episodes, starting from the source policy; the frozen control is reevaluated at these points without weight updates. The frozen source obtains 70.42% on the final test, the reference for the 11.56-point warm-start improvement. Training budgets match in episodes and PPO passes.

Table 13: Selected-checkpoint responses on C3: paired SR changes (percentage points).

Source	No task DDL	All at $t = 0$	Both
C1	+22.33	−13.75	+5.67
C2	+21.55	−20.21	+21.04
C3	+20.66	−17.75	+15.00
C4	+18.92	−23.83	+7.96
C5	+22.67	−8.21	+20.88

Table 14: Selected-checkpoint responses on C4: paired SR changes (percentage points).

Source	No task DDL	No subtask DDL	No outages	Both local
C1	+23.71	+1.96	+3.79	+2.21
C2	+21.88	−2.00	−1.58	−2.08
C3	+24.67	−2.12	+2.67	+4.38
C4	+17.12	−0.75	+2.21	+3.75
C5	+20.50	+0.00	−2.17	−1.21

Table 15: Selected-checkpoint responses on C5: paired SR changes (percentage points).

Source	No task DDL	Oracle rates	Oracle + no DDL	No fire	Reliable exec.
C1	+13.18	+3.48	+14.74	+0.00	+5.58
C2	+13.86	+1.92	+14.13	+0.00	+3.67
C3	+14.66	+3.35	+8.52	+0.83	+3.58
C4	+15.43	+2.35	+14.32	+1.25	+12.97
C5	+13.56	+0.10	+10.61	+2.00	+6.83

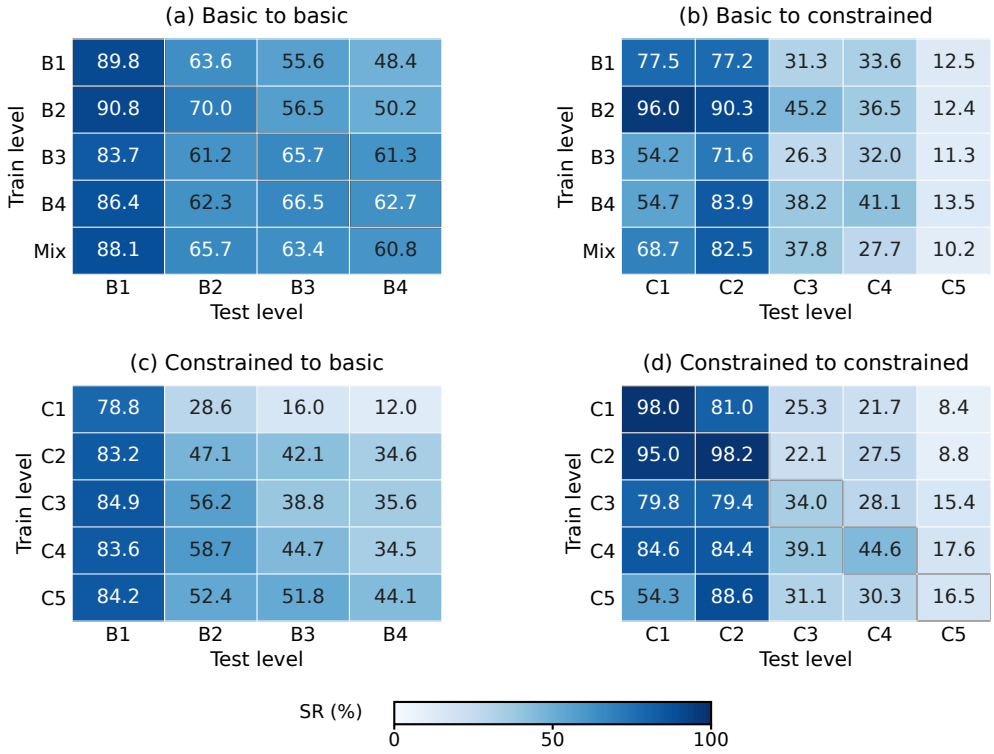


Figure 9: Complete frozen-policy transfer matrices on paired core cases. Rows are source training settings and columns are target levels. The shared color scale reports task success.

G Frozen Transfer to Minecraft and Factorio

Protocol. We transfer the C2 PPO checkpoint selected at update 120, the same checkpoint as the TaskBench experiment (Appendix F). We retain source-style categorical sampling and freeze all network weights. Domain adapters translate native tasks and execution states into the allocator's graph interface. Task completion requires native execution receipts. SR averages episode completion rates, with equal weight per configuration.

Minecraft configuration. Two task families cover farm production and factory crafting, each with six concurrent tasks and L2/L5 constraint settings. The farm uses four agents; the factory uses six. Native operations include collection, transport, processing, crafting, and placement. L2 provides 18 buffer slots and four output slots; L5 reduces these to 12 and two, hides operation success probabilities, and adds timed output collection and machine outages. Horizons range from 269 to 401 seconds. This extension also applies a bounded operation-call budget at L5.

Factorio configuration. Five production families, each with five constraint levels, define 25 configurations with 8–18 tasks. Native smelting, assembly, transport, and delivery share compatible machines and intermediate materials. The adapter exposes public resource-slot nodes; workers can carry 64 material units, and buffer chests use native stack capacity with one item type per chest. Initial raw materials cover the prescribed workload. Higher levels restrict compatibility and capacity, hide execution rates, and introduce deadlines and outages. Native inventory and delivery ledgers verify completed products.

Results. The frozen policy reaches 90% SR in Minecraft and 81% in Factorio (Table 16). Minecraft completes all tasks in both L2 configurations and the L5 factory; L5 farm completion drops to 58%. Factorio performance varies across levels, with L5 exceeding L4 rather than forming

Table 16: Frozen Overcooked PPO on native Minecraft and Factorio tasks.

Domain	Configuration	SR (%)
Minecraft	L2 factory	100
Minecraft	L2 farm	100
Minecraft	L5 factory	100
Minecraft	L5 farm	58
Minecraft	Overall	90
Factorio	L1	98
Factorio	L2	75
Factorio	L3	78
Factorio	L4	72
Factorio	L5	82
Factorio	Overall	81

a monotonic difficulty curve. These evaluations serve as representative feasibility checks of frozen deployment.