MAGIC: MIXED-GRANULARITY AGENT GRAPHS VIA INCREMENTAL CONSTRUCTION WITH DENSE-REWARD REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Collaboration topology determines the performance and execution cost of multi agent systems based on large language models. Existing representative topology generation methods typically adopt a single collaboration granularity, using either individual agents throughout the topology for fine grained control or predefined groups to reuse established collaboration patterns. However, even within the same task, different functional roles may require substantially different levels of collaboration. A fixed collaboration granularity therefore fails to capture the diverse collaboration demands across functional roles. Our key insight is to flexibly select the collaboration granularity for each functional role, thereby combining the fine grained control of individual agents with the reusable collaboration patterns of predefined groups. Introducing mixed granularity, however, exponentially enlarges the search space for topology generation. Existing topology generation methods typically rely only on final task outcomes as reinforcement learning rewards. Such sparse rewards provide insufficient intermediate feedback in the enlarged mixed granularity search space, which can lead to unstable convergence or slow topology generation. To address this challenge, we propose MAGIC, a dense reward reinforcement learning framework for mixed granularity graph generation. MAGIC constructs a mixed granularity MAS topology incrementally. For each functional role, it determines whether the role should be instantiated as an individual agent or a reusable group, and then connects the instantiated unit to the existing MAS topology. We directly optimize the construction policy using returns from trajectories sampled by the current policy. We further introduce a reward shaping mechanism based on structural complexity, structural redundancy, and utility. This mechanism provides intermediate feedback through sampled utility signals and structural signals while preserving the cumulative task reward. Experimental results show that MAGIC achieves the highest mean score among the evaluated methods across eight benchmarks and achieves strong inference efficiency in the efficiency evaluation.

1 INTRODUCTION

LLM-based multi-agent systems (MAS) coordinate specialized language-model agents through structured interaction (Wu et al., 2023; Du et al., 2023; Liu et al., 2023). Their effectiveness and cost depend on collaboration topology, which determines agent participation and information flow (Zhuge et al., 2024; Li et al., 2024; Zhang et al., 2024). Because tasks differ in complexity and required capabilities, recent work generates a topology for each task (Zhang et al., 2024; Li et al., 2026; Chen et al., 2026). The closest approaches take two forms: agent-level methods predict edges over specified agents or generate atomic-agent roles and connections (Zhang et al., 2024; Li et al., 2026), whereas group-level methods connect predefined collaborative groups as reusable units (Chen et al., 2026). Despite their different scales, both fix node granularity before construction, producing only atomic-agent or only group topologies.

As illustrated in Figure 1, this shared choice creates a *fixed-granularity mismatch*: subtasks within one query can demand different amounts of collaboration, yet every constructed unit must use the same organizational scale. Consider an API-update task.

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.

Investigating upstream changes and testing compatibility may merit a `SearchGroup` and a `TestingGroup`, while implementing a one- or two-line patch may need only a `ProgrammerAgent`. With fixed granularity, an all-atomic graph must assemble collaboration node by node, forgoing reusable group priors (Chen et al., 2026) and requiring longer construction sequences. An all-group graph treats each group as indivisible, limiting fine-grained structural control and assigning multiple agents to subtasks that one agent can handle. The controlled assignments in Section 3 (D1) show task- and role-dependent performance–cost preferences, motivating local granularity selection in a space that combines individual agents and reusable groups.

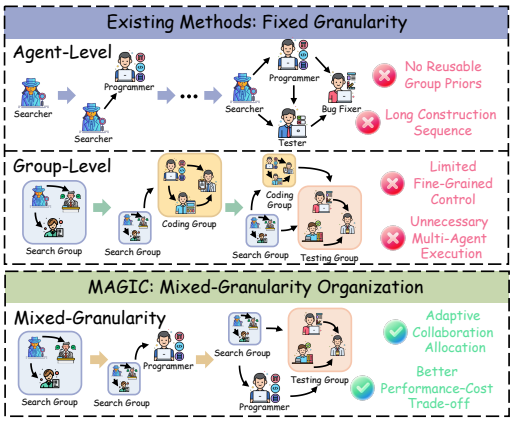


Figure 1: Fixed-granularity and MAGIC organizations for API updates.

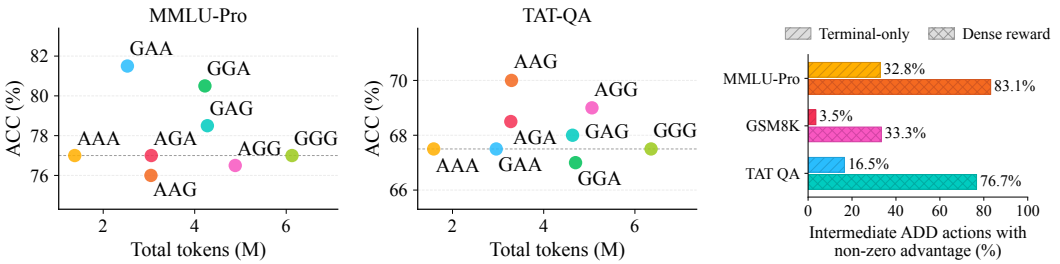
Learning in this expanded space challenges two optimization routes used by existing topology generators: search-based supervision and reinforcement learning. Search-then-SFT imitates construction trajectories selected through search and verification (Li et al., 2026; Chen et al., 2026). Mixed granularity enlarges the candidate space: even with a fixed role–edge skeleton, m dual-granularity roles admit 2^m realizations. D2 quantifies the fraction verifiable under a limited execution budget. Reinforcement learning instead directly optimizes topology generators using task rewards, as in G-Designer (Zhang et al., 2024). However, terminal rewards offer limited guidance for individual decisions in incremental construction. Trajectories sampled for the same query receive identical terminal-only returns when their final scores coincide, despite differences in their intermediate organizations. D3 shows that intermediate graph information distinguishes some such trajectories during learning. The matched-budget comparison in Section 5.4 further shows higher final-budget performance for dense-reward RL than terminal-only RL on both evaluated datasets. These results motivate direct policy optimization with construction-level feedback for mixed-granularity graphs.

We therefore propose MAGIC, a dense-reward reinforcement learning framework that generates mixed-granularity agent graphs incrementally from an empty graph. Given the task and partial graph, the policy selects a functional role, its atomic-or-group realization, and predecessors at each step to build an executable organization. These local granularity choices are learned from current-policy trajectories, without a pre-collected corpus of successful trajectories. We use potential-based reward shaping (Ng et al., 1999), which preserves the original task objective, to provide intermediate feedback from partial-graph task performance, structural complexity, and role repetition.

Our Contributions. (1) **Mixed-Granularity Perspective.** We formulate task-specific MAS topology generation in a mixed-granularity organization space that supports local selection between single agents and reusable groups, with all-atomic and all-group organizations as special cases. (2) **Dense-Reward Incremental Construction.** We propose MAGIC, which directly optimizes role, granularity, and dependency decisions using returns from current-policy trajectories with potential-based construction feedback. (3) **Experimental Evaluation.** MAGIC achieves the highest mean score among the evaluated methods across eight benchmarks and achieves a favorable inference performance–token trade-off in our efficiency study. Component ablations compare granularity choices under a shared expanded-node budget with comparable mean inference-call counts. Matched-budget SFT and terminal-only RL comparisons support construction-level feedback.

2 Problem Formulation

Given a task query $\mathcal{Q}$, we construct a task-specific multi-agent organization from a fixed domain library $\mathcal{R}$. We build its role profiles and admissible realizations once offline and hold them fixed across queries; construction selects the organization for $\mathcal{Q}$.



(a) (b) (c)

Figure 2: **Granularity preferences and construction feedback.** (a,b) D1: accuracy versus total input/output tokens on MMLU-Pro and TAT-QA, respectively. Letters indicate atomic (A) or group (G) decomposer, solver, and verifier realizations. TAT-QA uses strict-answer scoring. (c) D3: non-zero advantages at intermediate ADD actions, excluding initial and STOP actions.

Mixed-granularity organization space. Each library role $r \in \mathcal{R}$ admits one or both realizations in $\mathcal{Z}(r) \subseteq \{\mathsf{AtomicAgent}, \mathsf{Group}\}$. A task-specific mixed-granularity agent graph is

$$\begin{aligned} \mathcal{G} &= (\mathcal{U}, \mathcal{E}), \\ \mathcal{U} &= \{u_i = (i, r_i, z_i)\}_{i=1}^{N}, \quad z_i \in \mathcal{Z}(r_i), \end{aligned} \tag{1}$$

where $\mathcal{U}$ contains top-level units and $\mathcal{E} \subseteq \mathcal{U} \times \mathcal{U}$ their directed dependencies. An atomic unit executes its role directly; a group is a fixed internal agent graph with the same external interface. The mixed space $\mathbb{G}_{\text{mix}}$ contains the restricted all-atomic and all-group spaces $\mathbb{G}_{\text{atomic}}$ and $\mathbb{G}_{\text{group}}$.

Task objective and constraints. Executing organization $\mathcal{G}$ on query $\mathcal{Q}$ yields $\widehat{y} = \text{Execute}(\mathcal{Q}, \mathcal{G})$ (Section 4.1), scored as $S(\widehat{y}, y^*)$ against reference answer or tests y^*. We optimize policy-induced $p_\theta(\mathcal{G} \mid \mathcal{Q}, \mathcal{R})$ for expected task score:

$$\max_{\theta} \mathbb{E}_{(\mathcal{Q}, y^*)} \mathbb{E}_{\mathcal{G} \sim p_\theta(\cdot \mid \mathcal{Q}, \mathcal{R})} \left[S(\text{Execute}(\mathcal{Q}, \mathcal{G}), y^*) \right], \tag{2}$$

subject to $\mathcal{G} \in \mathbb{G}_{\text{legal}}$ and $N \leq N_{\max}$. Here $\mathbb{G}_{\text{legal}} \subseteq \mathbb{G}_{\text{mix}}$ requires admissible role realizations, compatible interfaces, and acyclic expanded graphs; $N_{\max}$ bounds the number of top-level units. Execution cost is evaluated alongside task performance. Gold answers or tests are used only after execution for scoring. We next examine local granularity preferences, configuration verification costs, and feedback for construction decisions.

3 Empirical Investigation

We study granularity preferences (D1), verification costs (D2), and construction feedback (D3).

D1: Granularity preferences are task and role dependent. Holding the library and execution settings fixed, we enumerate all eight atomic/group assignments of a serial decomposer–solver–verifier skeleton on 200 queries each from MMLU-Pro test and TAT-QA dev. Figure 2(a,b) shows that MMLU-Pro favors `GAA`, whereas TAT-QA favors `AAG`. These mixed assignments outperform both fixed-granularity endpoints by 4.5 and 2.5 accuracy points, respectively, while consuming fewer tokens than `GGG`. The preferred grouped role thus differs by task, supporting local granularity selection. Appendix D.1 gives the protocol and detailed comparisons.

D2: Verified configuration coverage is analytically bounded. A fixed role–edge skeleton with m dual-granularity roles has $|\mathcal{Z}_{\text{mix}}| = 2^m$ configurations. With a budget of B complete graph executions, the verifiable fraction satisfies

$$\text{Coverage}(B) \leq \min\left(1, \frac{B}{2^m}\right), \tag{3}$$

since each verified assignment costs at least one execution. Role and edge choices enlarge this space.

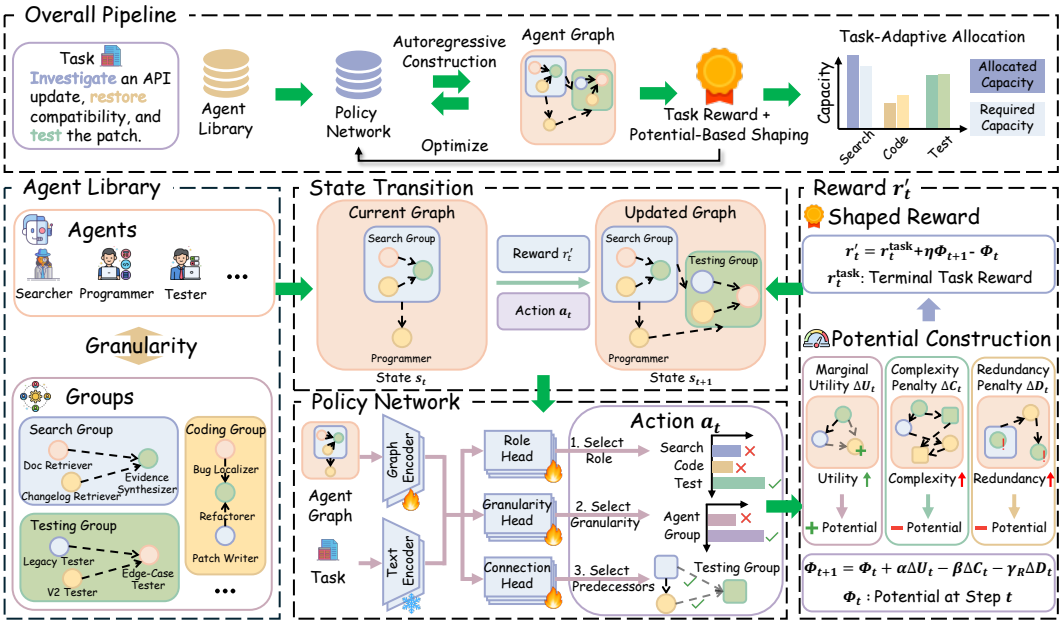


Figure 3: MAGIC constructs mixed-granularity agent graphs incrementally and trains the construction policy with potential-based dense rewards.

D3: Relative learning signals at intermediate construction steps. On the same 720 trajectories from MMLU-Pro, GSM8K, and TAT-QA, we compare terminal-only feedback with the dense reward defined in Section 4.3. Figure 2(c) shows higher non-zero advantage rates for intermediate ADD actions on all three datasets, including an increase from 32.81% to 83.13% on MMLU-Pro. Dense returns also distinguish some trajectories with equal terminal scores at intermediate positions. Appendix D.3 reports the protocol and full statistics; Sections 5.3 and 5.4 evaluate learning outcomes under fixed update counts with a shared expanded-node budget (Q2), and under matched API expenditure for training (Q3).

4 Methodology

Figure 3 summarizes MAGIC's graph construction, execution, and policy training.

4.1 Mixed-Granularity Realization and Execution

We instantiate the units defined in Section 2 with frozen LLM executors. An atomic unit uses one role executor; a group uses its internal agents and an aggregator that emits a single role-level result. Both realizations expose this result through the same input–output contract, allowing connected units to exchange role-level outputs regardless of granularity. Internal group membership and topology remain fixed during construction.

To execute a nonempty partial or final graph, we expand each group into its internal DAG and run the resulting atomic-agent graph in dependency order. A fixed outer summarizer produces the final answer from the public task, output requirements, and results from all sinks of the expanded graph. For the empty graph $\mathcal{G}_0$, including immediate STOP, the base LLM instead receives a minimal direct-answer prompt containing the question, any answer options, and the answer requirement. Appendices A.1 and C.7 detail execution and the role interfaces, respectively.

4.2 Task-Specific Incremental Construction

The policy observes the public query and partial agent graph. A frozen text encoder embeds the query and role realizations; node features combine realization semantics with granularity, construction po-

sition, and degree embeddings. An *Edge-aware GRU* encodes units in construction order, combining each unit's features with the mean representation of its actual predecessors and the query embedding. Its recurrent state captures construction history, whereas predecessor aggregation captures graph dependencies. Attention uses the query to pool the node states, and a projection combines the pooled representation, query, and construction progress into decision context $\mathbf{c}_t$. A learned empty-graph vector initializes this process. Appendix A.2 formalizes the encoder and action heads.

Three heads select the role, realization, and predecessors in sequence to parameterize a non-STOP action $a_t = (r_t, z_t, \mathbf{b}_t)$, where b_{jt} indicates an edge from existing unit j to the new unit:

$$\pi_\theta(a_t \mid s_t) = \pi_\theta(r_t \mid s_t)\pi_\theta(z_t \mid r_t, s_t) \prod_{j \in \mathcal{V}_t} \pi_\theta(b_{jt} \mid r_t, z_t, s_t). \tag{4}$$

Here $\mathcal{V}_t$ contains legal predecessors. Roles and STOP share one masked categorical distribution; a second categorical distribution selects the chosen role's realization. Independent Bernoulli decisions then select an admissible subset of predecessors. A STOP action uses only its role-level probability and skips the remaining heads. Legality masks enforce library, direction, and structural constraints.

Each addition updates the graph encoding before the next decision. Construction ends with an explicit STOP action. At the maximum unit horizon, or when the configured structural budget permits no addition, all other roles are masked, making STOP the only legal choice. At inference, the policy samples a graph with dropout disabled and then executes it; probe scores, gold answers, and rewards are training-only information. The text encoder and execution LLMs remain frozen throughout training; only the graph encoder, feature projections, pooling, and action heads are updated.

4.3 Potential-Based Dense-Reward Optimization

Task objective and graph potential. Let $S(\mathcal{G}, q) \in [0, 1]$ denote the score obtained by executing graph $\mathcal{G}$ on query q. Training maximizes expected terminal task score, using accuracy, official F1, or code-test success as appropriate. For each query's trajectory group, we select a fixed probe set P from other training queries and share it across all trajectories and construction steps. Executing partial graphs on these probes provides a common, multi-query reference for their utility, complementing the terminal score on the input query. Keeping P fixed lets us compare successive organizations on the same questions. Adding a unit changes the graph from $\mathcal{G}_t$ to $\mathcal{G}_{t+1}$, yielding the paired utility gain:

$$\Delta U_t = \frac{\sum_{p \in P}[S(\mathcal{G}_{t+1}, p) - S(\mathcal{G}_t, p)]}{|P| + \kappa}. \tag{5}$$

Here $\kappa \geq 0$ smooths the gain. We measure structural overhead by

$$C(\mathcal{G}) = w_n[N_{\text{exp}}(\mathcal{G}) - b_n]_+ + w_e[E_{\text{exp}}(\mathcal{G}) - b_e]_+,$$

where $[x]_+ = \max(0, x)$, N_{exp} and E_{exp} count nodes and edges after expanding groups, and b_n, b_e are free allowances. Role repetition is $D(\mathcal{G}) = \sum_r [m_r(\mathcal{G}) - 1]_+$, where m_r counts outer units assigned role r, regardless of granularity. Each addition contributes

$$d_t = \alpha \Delta U_t - \beta \Delta C_t - \gamma_R \Delta D_t,$$

where $\Delta C_t = C(\mathcal{G}_{t+1}) - C(\mathcal{G}_t)$ and ΔD_t is defined analogously. The weights balance utility gains, added structural complexity, and repeated roles in the shaping signal. Starting from $\Phi_0 = 0$, we accumulate $\Phi_{t+1} = \Phi_t + d_t$; fixed probes and consistent graph scores define the graph potential $\Phi_t = \Phi_P(\mathcal{G}_t)$ relative to the initial graph. Appendix A.3 details evaluation caching and the full reward definitions; Appendix C.6 lists the configuration.

Shaping and terminal settlement. Shaping and returns share discount η, giving reward

$$r'_t = r_t^{\text{task}} + \eta \Phi_{t+1} - \Phi_t. \tag{6}$$

The base reward is zero during construction and the task score at termination. We set initial and terminal potentials to zero. Thus, explicit STOP receives $S(\mathcal{G}, q) - \Phi_P(\mathcal{G})$, including when forced by the unit horizon. The implementation accumulates paired utility and structural increments to compute the intermediate potential. With the current $\eta = 1$, shaping redistributes reward across construction steps while the trajectory total remains its terminal score.

Policy optimization. For each query, we sample K trajectories from the current policy and compute discounted return-to-go $G_{k,t}$. Returns are normalized among trajectories that contain an action at the same position: $A_{k,t} = (G_{k,t} - \mu_t)/(\sigma_t + \epsilon)$. We minimize the action-averaged loss

$$\mathcal{L} = -\overline{\mathrm{sg}(A) \log \pi_\theta(a \mid s)} + \beta_{\mathrm{KL}} \overline{D_{\mathrm{KL}}(\pi_\theta \| \pi_{\mathrm{ref}})} - \beta_H \overline{H(\pi_\theta)}. \tag{7}$$

The bar averages over all recorded action positions, including the final STOP even when forced by the horizon. This probability-one action has zero log-probability but remains in position-aligned normalization and the averaging denominator. The operator sg stops gradients through advantages. The reference policy is a frozen copy from the start of RL. KL and entropy are computed over the full hierarchical action distribution at each visited state. This trains the construction policy directly from its own trajectories, without a pre-collected successful-trajectory corpus or a value network. Appendix A.3 specifies normalization and regularization.

Training costs include partial-graph probe evaluations and final-graph scoring, whereas inference executes only the completed organization. Probe evaluations and policy updates are confined to training; Appendix A.4 gives both procedures.

4.4 Policy Invariance of Reward Shaping

Following potential-based reward shaping (Ng et al., 1999), we establish the relation between the shaped and terminal task objectives. **Proposition 1 (Return preservation).** For a finite construction episode with T reward transitions, use the same discount η in shaping and returns, set $\Phi_0 = \Phi_T = 0$, and fully settle the potential under both sampled and forced termination. Then

$$\sum_{t=0}^{T-1} \eta^t r'_t = \sum_{t=0}^{T-1} \eta^t r_t^{\mathrm{task}}. \tag{8}$$

Consequently, shaping preserves the expected task return and its optimal policy set. In our undiscounted setting, both returns equal the final task score. The proof telescopes the potential differences and is given in Appendix B.

This property allows probe utility and structural information to guide intermediate construction decisions while leaving the terminal task objective unchanged. For $\eta = 1$, the shaped return-to-go is $G_{k,t} = S(\mathcal{G}_k, q) - \Phi_{k,t}$. The pre-action potential serves as a state-dependent reference for the organization already built. Shared probes provide a common utility reference across partial graphs, while structural terms summarize their complexity and role repetition. Together, these signals let training comparisons account for both terminal performance and the organization already built before each decision. Position-aligned normalization therefore compares terminal outcomes relative to these partial-graph potentials, yielding the intermediate signal differences examined in D3 even when final scores coincide. The proposition concerns environment returns; policy updates use the normalized, KL- and entropy-regularized loss in Equation 7. Sections 5.3 and 5.4 evaluate the learning outcomes of using these comparisons.

5 Experiments

We evaluate effectiveness (Q1), component contributions (Q2), optimization under matched training budgets (Q3), and inference efficiency (Q4).

5.1 Experimental Setup

Datasets. We evaluate MAGIC on eight benchmarks spanning four task families: (1) knowledge and commonsense reasoning: MMLU-Pro (Wang et al., 2024) and StrategyQA (Geva et al., 2021); (2) mathematical reasoning: AQuA (Ling et al., 2017) and GSM8K (Cobbe et al., 2021); (3) code generation: HumanEval (Chen et al., 2021) and LiveCodeBench-v6 (LCB-v6) (Jain et al., 2025); and (4) table and financial reasoning: TAT-QA (Zhu et al., 2021) and TabFact (Chen et al., 2019). Data and scoring protocols are detailed in Appendix C.2.

Baselines. We compare against 17 baselines in three categories: (1) direct/single-agent methods: DeepSeek V4 Flash and Qwen Plus; (2) established MAS: AutoGen (Wu et al., 2023) and

Table 1: Overall performance across eight benchmarks (%). We report mean ± standard deviation over three runs (seeds 41, 42, and 43), using official F1 for TAT-QA, pass@1 for code-generation tasks, and accuracy for all remaining tasks. The **best**, second-best, and third-best mean scores in each column are highlighted; tied scores share the same color.

Method	Knowledge & Commonsense (Acc. %)		Mathematical Reasoning (Acc. %)		Code Generation (pass@1 %)		Table & Financial Reasoning (F1 / Acc. %)	
	MMLU-Pro	StrategyQA	AQuA	GSM8K	HumanEval	LCB-v6	TAT-QA	TabFact
DeepSeek V4 Flash	72.53 ±0.61	85.73 ±0.12	90.03 ±1.27	93.78 ±1.35	84.35 ±0.35	20.19 ±0.87	70.45 ±0.15	92.20 ±0.35
Qwen Plus	60.33 ±0.31	77.80 ±0.00	68.90 ±0.68	58.91 ±0.27	91.06 ±0.70	27.24 ±0.66	69.52 ±0.32	80.53 ±0.50
AutoGen	73.00 ±0.72	84.00 ±0.20	89.37 ±0.79	92.37 ±0.61	82.93 ±0.00	18.86 ±0.57	70.28 ±0.43	92.47 ±0.61
LLM-Debate	72.27 ±0.12	84.20 ±0.35	89.24 ±0.91	92.19 ±0.27	84.76 ±0.00	21.52 ±0.33	70.62 ±0.48	92.87 ±0.31
DyLAN	73.47 ±0.58	82.93 ±0.42	89.90 ±0.91	93.20 ±0.38	85.57 ±0.93	22.29 ±0.00	70.31 ±0.09	94.40 ±0.87
GPTSwarm	70.93 ±0.81	84.27 ±0.31	90.94 ±0.79	94.39 ±0.60	84.35 ±0.70	19.62 ±0.33	70.26 ±0.57	92.27 ±0.12
G-Designer	70.27 ±0.70	87.53 ±0.58	89.50 ±1.20	93.81 ±0.49	85.37 ±0.61	22.86 ±0.00	70.10 ±0.37	91.20 ±1.04
Sparse-Comm	70.33 ±0.64	86.33 ±0.23	88.71 ±0.45	93.25 ±1.44	84.35 ±0.35	21.52 ±0.33	70.52 ±0.30	91.67 ±0.23
GraphSearch	71.00 ±0.72	86.47 ±0.31	89.50 ±0.99	93.45 ±1.06	80.08 ±0.35	21.90 ±0.33	70.38 ±0.35	90.53 ±0.50
Graph-R1	72.00 ±0.40	84.40 ±0.87	89.24 ±0.23	93.91 ±0.49	86.18 ±0.35	19.43 ±0.00	69.48 ±0.19	91.27 ±0.42
R-GFM	68.27 ±0.70	84.00 ±1.25	88.85 ±0.45	94.26 ±0.04	88.01 ±0.35	18.86 ±0.00	70.45 ±0.87	90.67 ±0.99
BIGMAS	68.67 ±0.12	84.40 ±0.72	89.63 ±0.45	93.78 ±0.53	83.54 ±0.00	22.10 ±0.66	70.62 ±0.81	91.20 ±0.20
MasRouter	71.27 ±0.81	84.27 ±0.58	88.85 ±0.23	94.14 ±0.09	75.61 ±0.00	21.33 ±0.33	70.81 ±0.49	91.53 ±0.31
GoAgent	70.60 ±0.20	83.93 ±0.23	89.11 ±0.60	93.76 ±0.80	80.08 ±0.35	20.00 ±0.57	70.17 ±0.23	92.60 ±0.35
VeriMAP	72.33 ±0.64	82.60 ±0.35	89.90 ±0.82	93.48 ±0.86	78.25 ±0.70	20.95 ±0.66	70.21 ±0.31	93.27 ±0.12
ARG-Designer	73.20 ±0.53	83.60 ±0.60	91.86 ±0.23	93.40 ±0.95	89.43 ±0.93	18.67 ±0.33	71.01 ±1.06	92.33 ±1.10
EIB-Learner	71.87 ±0.46	84.20 ±0.72	90.42 ±0.60	92.95 ±1.51	82.93 ±0.00	21.52 ±0.33	69.91 ±0.12	92.67 ±0.23
MAGIC (Ours)	**78.27** ±1.90	**88.73** ±0.76	**93.70** ±1.42	**94.87** ±1.19	**94.72** ±1.96	**36.57** ±0.57	**71.80** ±0.25	**96.07** ±0.64

LLM-Debate (Du et al., 2023); and (3) adaptive/learned organizations: DyLAN (Liu et al., 2023), GPTSwarm (Zhuge et al., 2024), G-Designer (Zhang et al., 2024), Sparse-Comm (Li et al., 2024), GraphSearch (Wu et al., 2024), Graph-R1 (Luo et al., 2025), R-GFM (Liu et al., 2026), BIGMAS (Hao et al., 2026), MasRouter (Yue et al., 2025), GoAgent (Chen et al., 2026), VeriMAP (Xu et al., 2026), ARG-Designer (Li et al., 2026), and EIB-Learner (Shen et al., 2025). Baseline evaluation protocols and implementation settings appear in Appendices C.3 and C.6.

5.2 Q1: Effectiveness

For Q1, MAGIC uses DeepSeek V4 Flash as its executor. We train a separate construction policy for each benchmark on 100 training examples; HumanEval uses the same 100 examples as LiveCodeBench. For each benchmark, we repeat the complete training and evaluation procedure three times with random seeds 41, 42, and 43, evaluating the final checkpoint from each run. We report the mean and standard deviation across the three runs on a fixed data partition. Training and utility probes use only the designated training subset, disjoint from the evaluation subset. MMLU-Pro and StrategyQA use custom held-out partitions; source splits, sample counts, and allocation rules are detailed in Appendix C.3.

To answer **Q1**, Table 1 compares MAGIC with 17 baselines across four task families. MAGIC achieves the highest mean score on all eight benchmarks, outperforming direct inference, established MAS, and adaptive organizations. The largest gains over the strongest baselines occur on LCB-v6 and MMLU-Pro: pass@1 increases from 27.24% to 36.57% (9.33 percentage points), while accuracy rises from 73.47% to 78.27% (4.80 points). On HumanEval, MAGIC improves pass@1 from 91.06% to 94.72%, a further 3.66-point gain. The advantage also extends to mathematical and structured-data reasoning, with gains over the strongest baselines of 1.84 points on AQuA and 1.67 points on TabFact. On GSM8K and TAT-QA, where the leading baselines obtain closely clustered scores, the mean gains are smaller at 0.48 accuracy points and 0.79 F1 points, respectively. The gains span reasoning accuracy and executable-code correctness, with magnitudes varying by benchmark.

The leading baseline varies across tasks, highlighting the importance of task-dependent organization. Qwen Plus outperforms every MAS baseline on both code-generation benchmarks, yet MAGIC improves further on both. On MMLU-Pro and TAT-QA, it surpasses the learned organizations DyLAN and ARG-Designer, respectively; on StrategyQA, it exceeds G-Designer by 1.20 accuracy points. This pattern is consistent with the motivation for adapting collaboration to each task. MAGIC

Table 2: Component ablations under a shared expanded-node budget. Calls and tokens are per-question means; tokens are in thousands (k). A/G denotes mean outer atomic/group counts, excluding the outer summarizer. Each variant uses one run with seed 23.

Variant	MMLU-Pro				TAT-QA			
	Acc. (%)	Calls	Tokens (k)	A/G	F1 (%)	Calls	Tokens (k)	A/G
All-Atomic	72.00	8.10	11.80	6.80/0.00	80.00	8.80	13.20	6.90/0.00
All-Group	74.00	8.30	12.60	0.00/1.75	81.00	9.00	14.10	0.00/1.80
Initialization only	68.00	8.20	12.40	2.20/1.20	78.00	8.90	13.80	2.00/1.25
Terminal-only RL	75.00	8.40	12.10	2.30/1.20	81.00	9.10	13.70	2.10/1.25
MAGIC (Full)	76.00	8.20	11.60	2.20/1.20	81.50	8.90	13.30	2.20/1.20

supports such adaptation through local atomic-or-group choices and feedback-driven construction learning. The ablations examine organization-space and training contributions.

5.3 Q2: Component Ablations

To answer **Q2**, we compare five variants on MMLU-Pro and TAT-QA with DeepSeek V4 Flash and $N_A + 4N_G \leq 8$ expanded execution nodes. Outer-unit and expanded-depth limits are both eight. All-Atomic and All-Group restrict granularity; Initialization only uses untrained weights; Terminal-only RL removes shaping. Full retains mixed granularity and dense feedback. This Q2-specific configuration and its data partitions are detailed in Appendix C.4.

Table 2 shows that Full obtains the highest scores on both datasets with comparable mean call counts: 8.10–8.40 on MMLU-Pro and 8.80–9.10 on TAT-QA. Relative to All-Atomic, Full gains 4.00 accuracy points and 1.50 F1 points, using 1.69% fewer tokens on MMLU-Pro and 0.76% more on TAT-QA. Relative to All-Group, it gains 2.00 and 0.50 points while reducing tokens by 7.94% and 5.67%, respectively. Full also improves on Initialization only by 8.00 and 3.50 points with fewer tokens. Compared with Terminal-only RL, dense feedback adds 1.00 and 0.50 points and reduces tokens by 4.13% and 2.92%. Full averages 2.20 atomic units and 1.20 groups per query on each dataset. These results support its performance–cost advantage under the evaluated constraint; the A/G means summarize aggregate usage, without identifying the fraction of individual graphs that mix granularities.

5.4 Q3: Optimization Strategies

To answer **Q3**, we compare Search-then-SFT (SFT), Terminal-only RL, and Dense-reward RL under matched DeepSeek V4 Flash API training budgets. SFT imitates the highest-scoring positive candidates obtained through search and verification. Both RL variants update from current-policy trajectories: Terminal-only RL uses final task scores, while Dense-reward RL adds potential-based construction feedback. At each budget stage, all three methods share 200 held-out queries per dataset, scored by MMLU-Pro accuracy and TAT-QA official F1 (Appendix C.5).

Figure 4 shows that Dense-reward RL outperforms Search-then-SFT at four of the five nonzero budget stages on each dataset. At CNY 10, it reaches 78.00% accuracy on MMLU-Pro and 74.95% F1 on TAT-QA, exceeding Search-then-SFT by 8.00 and 2.35 percentage points, respectively. Terminal-only RL reaches 75.00% and 74.34%, respectively, also exceeding SFT at the final budget. Dense feedback adds a further 3.00 accuracy points and 0.61 F1 points over terminal-only training.

The three-way comparison supports direct policy optimization and the additional benefit of construction-level feedback at

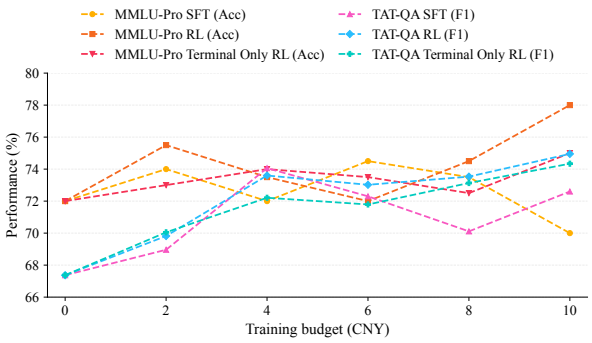


Figure 4: Performance under DeepSeek V4 Flash API budgets for Search-then-SFT (SFT), Terminal-only RL, and Dense-reward RL (RL).

the final budget. Dense-reward RL leads
Terminal-only RL at three of five nonzero
budget stages on MMLU-Pro and four on TAT-QA, with the largest final-budget gain on MMLU-Pro. Whereas Q2 fixes trajectory and update counts under a common expanded-node budget, Q3 compares performance under matched API expenditure, accounting for the execution cost of training feedback. The two experiments use different evaluation sets and graph limits; their absolute scores cannot be compared directly across the two protocols.

5.5 Q4: Efficiency

To answer **Q4**, we compare performance and total inference-token consumption with alternative fixed and learned agent organizations on four representative benchmarks, reusing Q1 performance results for overlapping methods. Performance and total tokens are averaged over seeds 41, 42, and 43. We count input and output tokens across all execution calls, including group-internal work and final aggregation; training costs are accounted for separately.

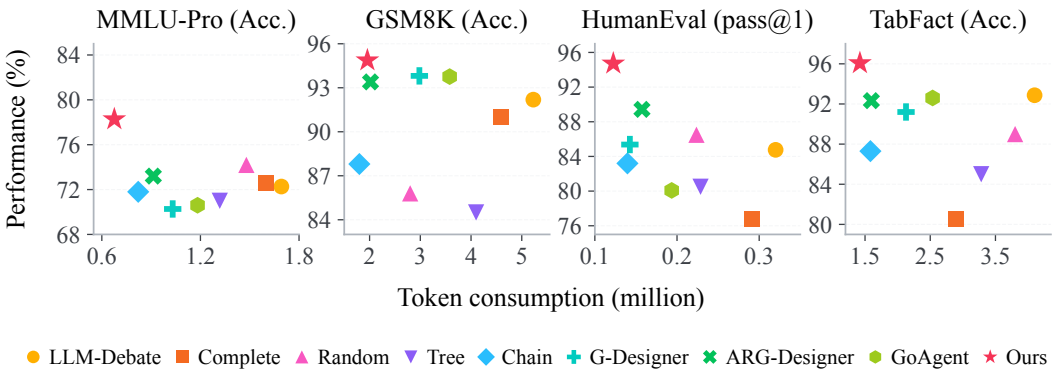


Figure 5: Performance–token trade-off across four datasets. The x-axis reports total inference tokens per evaluation set, and the y-axis reports task performance (accuracy or pass@1), both averaged over three seeds. Points toward the upper left indicate a more favorable trade-off.

Figure 5 shows that MAGIC lies on the Pareto frontier among the compared organizations on all four datasets, achieving the highest performance throughout and the lowest token consumption on MMLU-Pro, HumanEval, and TabFact. Relative to the highest-performing baseline on each of these three datasets—Random, ARG-Designer, and LLM-Debate, respectively—MAGIC improves performance by 4.07, 5.29, and 3.20 percentage points while reducing token consumption by 54.31%, 22.17%, and 65.33%. On GSM8K, it achieves 94.87% accuracy, exceeding G-Designer by 1.06 percentage points with 34.37% fewer tokens. Compared with Chain, the lowest-token organization on GSM8K, MAGIC gains 7.07 accuracy points with 8.80% more tokens. These results show a favorable performance–token trade-off across the evaluated task families, consistent with the mixed-granularity design that introduces collaboration locally while retaining single-agent execution for other units.

6 Conclusion

MAGIC formulates MAS design as mixed-granularity construction, locally realizing each role as an atomic agent or reusable group. Its policy learns from current-policy trajectories through potential-shaped returns and regularized policy gradients, without a pre-collected successful trajectory corpus. Shaping provides intermediate feedback while preserving the task reward. Across four task families and 17 baselines, MAGIC achieves the highest mean score on all eight benchmarks. It lies on the performance–token Pareto frontier on four benchmarks and uses the fewest tokens on three among the organizations included in the efficiency comparison. Under Q2's expanded-node budget, Full outperforms both fixed-granularity variants on MMLU-Pro and TAT-QA with comparable mean inference-call counts. Dense feedback also improves final-budget performance over terminal-only RL on MMLU-Pro and TAT-QA, supporting construction-level feedback alongside local granularity selection for task-adaptive collaboration.

AI USE STATEMENT

Generative AI tools were used during the preparation of this manuscript for limited purposes, including language refinement, literature retrieval and discovery, and research ideation support. Specifically, AI tools assisted with improving writing clarity, organizing related work, exploring potential research directions, and discussing possible methodological designs and experimental considerations.

REPRODUCIBILITY STATEMENT

We provide an anonymized reference implementation of MAGIC, together with the main-method experiment configuration, training and evaluation scripts, and validation tests at `https://anonymous.4open.science/r/MAGIC-1234/`. The main paper describes the empirical diagnostic protocols, mixed-granularity graph construction, policy optimization with dense rewards, and evaluation settings and results. The appendix provides detailed algorithms, theoretical derivations, experiment-specific protocols, hyperparameters, role libraries, and execution interfaces. The code package excludes datasets, model weights, checkpoints, and experiment outputs, but documents the required data formats and source splits, external model dependencies, sandbox setup, and instructions for running the main-method training and evaluation pipeline.

REFERENCES

Hongjiang Chen, Xin Zheng, Yixin Liu, Pengfei Jiao, Shiyuan Li, Huan Liu, Zhidong Zhao, Ziqi Xu, Ibrahim Khalil, and Shirui Pan. Goagent: Group-of-agents communication topology generation for llm-based multi-agent systems. *arXiv preprint arXiv:2603.19677*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. Tabfact: A large-scale dataset for table-based fact verification. *arXiv preprint arXiv:1909.02164*, 2019.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Mor Geva, Daniel Khashabi, Elad Segal, Tushar Khot, Dan Roth, and Jonathan Berant. Did aristotle use a laptop? a question answering benchmark with implicit reasoning strategies. *Transactions of the Association for Computational Linguistics*, 9:346–361, 2021.

Guangfu Hao, Yuming Dai, Xianzhe Qin, and Shan Yu. Brain-inspired graph multi-agent systems for llm reasoning. *arXiv preprint arXiv:2603.15371*, 2026.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Shiyuan Li, Yixin Liu, Qingsong Wen, Chengqi Zhang, and Shirui Pan. Assemble your crew: Automatic multi-agent communication topology design via autoregressive graph generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 23142–23150, 2026.

Yunxuan Li, Yibing Du, Jiageng Zhang, Le Hou, Peter Grabowski, Yeqing Li, and Eugene Ie. Improving multi-agent debate with sparse communication topology. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 7281–7294, 2024.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 158–167, 2017.

Haokun Liu, Zezhong Ding, and Xike Xie. Learning graph foundation models on riemannian graph-of-graphs. *arXiv preprint arXiv:2605.09993*, 2026.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic llm-powered agent network for task-oriented agent collaboration. *arXiv preprint arXiv:2310.02170*, 2023.

Haoran Luo, Guanting Chen, Qika Lin, Yikai Guo, Fangzhi Xu, Zemin Kuang, Meina Song, Xiaobao Wu, Yifan Zhu, Luu Anh Tuan, et al. Graph-r1: Towards agentic graphrag framework via end-to-end reinforcement learning. *arXiv preprint arXiv:2507.21892*, 2025.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pp. 278–287. Citeseer, 1999.

Xu Shen, Yixin Liu, Yiwei Dai, Yili Wang, Rui Miao, Yue Tan, Shirui Pan, and Xin Wang. Understanding the information propagation effects of communication topologies in llm-based multi-agent systems. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 12358–12372, 2025.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290, 2024.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Xixi Wu, Yifei Shen, Caihua Shan, Kaitao Song, Siwei Wang, Bohang Zhang, Jiarui Feng, Hong Cheng, Wei Chen, Yun Xiong, et al. Can graph learning improve planning in llm-based agents? *Advances in Neural Information Processing Systems*, 37:5338–5383, 2024.

Tianyang Xu, Dan Zhang, Kushan Mitra, and Estevam Hruschka. Verification-aware planning for multi-agent systems. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7528–7546, 2026.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. Masrouter: Learning to route llms for multi-agent systems. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15549–15572, 2025.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks. *arXiv preprint arXiv:2410.11782*, 2024.

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. Tat-qa: A question answering benchmark on a hybrid of tabular and textual content in finance. In *Proceedings of the 59th annual meeting of the Association for Computational Linguistics and the 11th international joint conference on natural language processing (volume 1: long papers)*, pp. 3277–3287, 2021.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.

A Detailed Method and Algorithms

This section expands the construction policy, executable mixed-granularity agent graph, and training procedure of MAGIC in Section 4. We first specify how the fixed library is executed, then describe the policy computation, potential-based rewards, and policy updates. Numerical settings describe the default configuration unless stated otherwise. Q2 overrides the graph limits and embedding capacity as specified in Appendix C.4; diagnostic-specific settings are reported with their respective protocols.

A.1 Role Realizations and Agent Graph Execution

A domain library supplies role profiles and their admissible atomic and group realizations. Each selected realization occupies one outer unit $u_i = (i, r_i, z_i)$, regardless of its internal size. An atomic realization uses one role executor. In the current group templates, three workers feed an internal aggregator, which emits the group's role-level result. For example, the calculation-auditor group combines a primary calculator, an independent recalculator, and a magnitude-and-unit checker with a calculation aggregator. Group membership, internal edges, role prompts, and tool permissions are fixed by the library; the policy chooses the outer roles, realizations, dependencies, and stopping position.

Outer edges point from existing units to newly appended units. During execution, groups expand into their fixed internal DAGs and agents run in dependency order. Each node request includes the public question, context, answer-format requirements, and a `predecessors` JSON array. Its structured message objects retain predecessor identifiers, roles, granularities, status, and payload fields such as analysis, answers, and tool observations. The request is serialized as JSON text for the execution LLM. Message projection and length budgets bound the supplied content.

All three workers in a group receive the results of every external direct predecessor. The internal aggregator receives the three workers as its direct predecessors, together with the public task, answer requirements, and role description. External predecessor messages are not additionally routed directly to this aggregator. The group exposes only the aggregator's result. The fixed outer summarizer receives the original question, output requirements, and results from all sinks of the expanded graph. This execution rule applies to nonempty partial graphs used for probes as well as completed graphs. An empty graph instead invokes the base LLM with the question, available answer options, and answer requirement, as defined in Section 4.1.

The construction policy observes the public query and graph structure. Executor responses, probe scores, and gold answers do not enter its decision context. At inference, construction is completed before the organization is executed. Tool requests and their results are handled by the execution layer; training updates neither the tools nor the execution LLM. The current execution configuration uses temperature zero, disabled thinking, `json_object` output, and a 2,048-token output limit. Core response content includes analysis and an answer, with field validity handled by the execution parser. The executor backbone and full prompts are experiment-specific.

A.2 Task-Conditioned Construction Policy

Frozen semantic features. The `sentence-transformers/all-MiniLM-L6-v2` encoder produces 384-dimensional semantic vectors. The public task view separates the question and options, public context, and metadata such as dataset and role/tool information. Long fields are chunked at the tokenizer limit. Each chunk is L2-normalized; chunk vectors are averaged and normalized again to obtain a field vector. With missing fields omitted, the task embedding is

$$\mathbf{q} = \mathrm{Normalize}\left(0.5\mathbf{e}_{\mathrm{question}} + 0.4\mathbf{e}_{\mathrm{context}} + 0.1\mathbf{e}_{\mathrm{metadata}}\right). \tag{9}$$

Answer-format instructions are supplied to the executors separately. Task and role-realization embeddings are cached and frozen during training.

Graph features and recurrence. Each outer unit has a 256-dimensional feature vector

$$\mathbf{x}_i = W_s\mathbf{e}_i + \mathbf{e}_{z_i} + \mathbf{e}_{\mathrm{pos}(i)} + \mathbf{e}_{\deg^-(i)} + \mathbf{e}_{\deg^+(i)}, \tag{10}$$

where $\mathbf{e}_i$ is its frozen realization embedding. The semantic projection, granularity embedding, construction-position embedding, and outer-degree embeddings are trainable. Position, indegree, and outdegree each use a separate $N_{\max} \times 256$ embedding table: 3×256 in Q1 and Q3, and 8×256 in Q2. Construction positions range from 0 to $N_{\max} - 1$; degrees are measured on the outer agent graph. Unsupported indices raise an error rather than being clamped. A shared Edge-aware GRUCell processes units in construction order:

$$\begin{aligned} \mathbf{p}_i &= \mathrm{Mean}_{j \to i}\, \mathbf{h}_j, & \mathbf{v}_i &= W_x \mathbf{x}_i + W_p \mathbf{p}_i + W_q \mathbf{q}, \\ \mathbf{u}_i &= \mathrm{GRUCell}(\mathrm{Dropout}(\mathbf{v}_i), \mathbf{u}_{i-1}), & \mathbf{h}_i &= \mathrm{LayerNorm}(\mathbf{u}_i). \end{aligned} \tag{11}$$

An empty predecessor set gives $\mathbf{p}_i = \mathbf{0}$, and $\mathbf{u}_0 = \mathbf{0}$. The recurrent state represents construction history; predecessor aggregation represents actual outer dependencies. Both belong to the policy's structural encoding. Agent-to-agent execution messages continue to follow the selected edges. The current graph is re-encoded at every construction decision, including its updated degrees.

Pooling and decision context. Task-conditioned attention pools the node representations:

$$a_i = \mathrm{softmax}_i \left(\frac{(W_k \mathbf{h}_i)^\top W_a \mathbf{q}}{\sqrt{256}} \right), \qquad \mathbf{g}_{\mathrm{pool}} = \sum_i a_i \mathbf{h}_i. \tag{12}$$

For an empty graph, a learned 256-dimensional vector replaces $\mathbf{g}_{\mathrm{pool}}$. Concatenating the pooled vector with $\mathbf{q}$ and applying a linear projection and LayerNorm gives a 256-dimensional graph vector $\mathbf{g}_t$. The action context is

$$\mathbf{c}_t = \mathrm{LayerNorm}\left(\mathrm{GELU}\left(W_c[\mathbf{g}_t; \mathbf{q}; N_t/N_{\max}; 1 - N_t/N_{\max}] + \mathbf{b}_c\right)\right), \tag{13}$$

where N_t is the current outer-unit count and W_c maps 642 inputs to 256 outputs.

Hierarchical actions. A shared role-scoring MLP receives $\mathbf{c}_t$ and each role's semantic embedding; a separate MLP scores STOP from $\mathbf{c}_t$. These scores enter one masked categorical distribution. Conditional on the sampled role, the granularity MLP receives the context, role semantics, and atomic and group semantics to score the admissible realizations. Conditional on both choices, the connection head forms a query from their semantics and context, and keys from existing node states. Each edge score combines a query–key dot product and an interaction MLP over the query, node state, and their coordinate-wise product. Legal predecessor indicators are sampled as independent Bernoulli variables. The current head dimensions are:

Module	Dimensions and transformation
Role MLP	$640 \to 256 \to 1$, intermediate GELU; shared across roles
STOP MLP	$256 \to 256 \to 1$, intermediate GELU
Granularity MLP	$1408 \to 256 \to 2$, intermediate GELU
Connection query	$1024 \to 256$, linear with bias
Connection key	$256 \to 256$, linear without bias
Connection interaction MLP	$768 \to 256 \to 1$, intermediate GELU

For connection query $\mathbf{v}_{r,z}$ and key $\mathbf{k}_j$ obtained from node state $\mathbf{h}_j$, the edge score is

$$\ell_{r,z,j} = \mathbf{v}_{r,z}^\top \mathbf{k}_j + \mathrm{MLP}([\mathbf{v}_{r,z}; \mathbf{h}_j; \mathbf{v}_{r,z} \odot \mathbf{h}_j]). \tag{14}$$

The connection dot product is unscaled, and its interaction term uses the node state $\mathbf{h}_j$. Task-conditioned graph pooling uses the $1/\sqrt{256}$ scaling in Equation 12. Decision heads omit dropout.

Writing $p_{j,t}$ for the probability of predecessor edge j, the sampled non-STOP action has log-probability

$$\begin{aligned} \log \pi_\theta(a_t \mid s_t) = {} & \log \pi_\theta(r_t \mid s_t) + \log \pi_\theta(z_t \mid r_t, s_t) \\ & + \sum_{j \in \mathcal{V}_t} \left[b_{j,t} \log p_{j,t} + (1 - b_{j,t}) \log(1 - p_{j,t}) \right]. \end{aligned} \tag{15}$$

Both selected and absent legal edges contribute. A sampled STOP uses only its role-layer probability. The first unit has no existing predecessor, so its edge term is an empty sum. Masks restrict choices to available roles and realizations, compatible interfaces, and the expanded-depth bound; Q2 additionally imposes the expanded-node budget in Appendix C.4. Predecessors must be existing units and edges follow construction order, with self-loops and duplicate predecessor entries excluded. Zero-predecessor additions remain legal after the first unit, allowing independent branches. Repeated roles are also legal and contribute to the role-repetition component of the shaping potential. No separate indegree cap is imposed.

Expanded depth counts nodes on the longest path, with root depth one. It includes group workers and internal aggregators and excludes the outer summarizer. An independent atomic unit has depth one, an independent group has depth two, and two serial groups have depth four. STOP is always legal. At the outer-unit limit, or when a configured budget leaves no legal addition, all non-STOP roles are masked, leaving a probability-one STOP action. This action remains an explicit position in the trajectory.

The policy uses one shared GRUCell with hidden size 256, dropout 0.1 on each GRU input projection sum during training, and inference policy sampling temperature 1.0. Q1 and Q3 allow at most three outer units and expanded depth four; Q2 uses eight for both limits, with at most eight expanded execution nodes. Evaluation disables dropout and retains probabilistic graph sampling. The policy starts from random initialization without STOP warmup. Semantic projections, structural embeddings, the GRU, attention pooling, context projection, and action heads are trainable; MiniLM and the execution LLM remain frozen.

A.3 Potential-Based Rewards and Policy Optimization

Probe utility and evaluation reuse. Four probe questions are selected without replacement from the remaining training pool using a reproducible, seed-dependent pseudorandom hash ordering, and are held fixed throughout each trajectory group as the probe set P. The current training query is excluded. Candidate hash keys depend on the experiment seed, current query ID, update index, and data-source version; the first four candidates in this ordering are selected. Probe questions may recur across updates. Each partial graph is executed with its selected roles and edges on the probe questions, measuring utility on shared training queries. The terminal score evaluates the completed graph on the conditioning query. For score $S(\mathcal{G}, p) \in [0, 1]$, define

$$U_P(\mathcal{G}) = \frac{\kappa u_0 + \sum_{p \in P} S(\mathcal{G}, p)}{\kappa + |P|}. \tag{16}$$

Parent–child differences cancel the shared prior:

$$\Delta U_P = \frac{\sum_{p \in P} [S(\mathcal{G}', p) - S(\mathcal{G}, p)]}{\kappa + |P|}. \tag{17}$$

The evaluation ledger stores reusable graph–query execution results. A missing graph–probe pair is executed and recorded; an available reusable record is read directly. Probe membership stays fixed independently of cache availability. Terminal and probe executions populate the ledger according to their execution paths. These records support reward computation, not policy inputs.

Structural potential. Let N_{exp} and E_{exp} count execution nodes and edges after group expansion. With $[x]_+ = \max(0, x)$, define

$$\begin{aligned} C(\mathcal{G}) &= w_n [N_{\text{exp}}(\mathcal{G}) - b_n]_+ + w_e [E_{\text{exp}}(\mathcal{G}) - b_e]_+, \\ D(\mathcal{G}) &= \sum_{r \in \mathcal{R}} [m_r(\mathcal{G}) - 1]_+, \end{aligned} \tag{18}$$

where m_r counts outer units assigned role r across both realizations. For every addition, the accumulated potential changes by

$$d_t = \alpha \Delta U_P - \beta \Delta C - \gamma_R \Delta D, \qquad \Phi_{t+1} = \Phi_t + d_t, \quad \Phi_0 = 0. \tag{19}$$

With fixed probes and consistent recorded evaluations, this accumulation implements the initial-graph-relative potential defined in Section 4.3.

Rewards and termination. An ordinary addition receives shaping reward $F_t = \eta\Phi_{t+1} - \Phi_t$. Every trajectory ends with an explicit STOP action, including when the outer-unit limit makes STOP the only legal choice. At this position, the graph receives its task score and the potential is fully settled to zero. The trajectory stores addition rewards in `transitions`, all addition and STOP decisions in `actions`, and terminal scoring and settlement separately as `terminal_reward` and `terminal_shaping_reward`. The settlement equals the negative pre-STOP potential and is retained even when negative.

Let T_k be the STOP action index. Return computation combines the two terminal fields at this position and then proceeds backward:

$$\begin{aligned} G_{k,T_k} &= r_{\text{terminal},k} + r_{\text{terminal_shaping},k} = S(\mathcal{G}_k, q) - \Phi_{k,T_k}, \\ G_{k,t} &= r_{k,t}^{\text{dense}} + \eta G_{k,t+1} \quad (t < T_k). \end{aligned} \tag{20}$$

The terminal terms are kept separate from the final addition's stored dense reward. Immediate STOP has zero initial potential and receives the direct-answer score. A horizon-forced STOP has log-probability zero but still occupies an action position for return normalization and loss averaging.

Terminal scores use accuracy for ordinary answer tasks, official F1 for TAT-QA, and single-sample test success for coding, reported as pass@1. The current call and token reward weights are zero. Reward parameters are $\alpha = 1$, $\beta = 0.02$, $\gamma_R = 0.05$, $\eta = 1$, $\kappa = 1$, $u_0 = 0.5$, $w_n = w_e = 1$, and $b_n = b_e = 6$. The free allowances apply to expanded nodes and edges. Structural terms redistribute intermediate credit; full terminal settlement leaves the trajectory's total reward equal to its task score.

Position-aligned returns and advantages. For the Q1 main configuration, $K = 8$ trajectories are sampled for each query; Q2 and Q3 use $K = 2$ under their respective protocols. Let $L_k = T_k + 1$ count recorded action positions in trajectory k, including its final STOP even when forced by the horizon. After incorporating terminal scoring and settlement, compute discounted returns backward with the same η used for shaping. For $\eta = 1$, the return at an actual action position is

$$G_{k,t} = S(\mathcal{G}_k, q) - \Phi_{k,t}, \tag{21}$$

where $\mathcal{G}_k$ is final and $\Phi_{k,t}$ its pre-action potential. Set $I_t = \{k : t < L_k\}$ and $n_t = |I_t|$; compute

$$\begin{aligned} \mu_t &= \frac{1}{n_t}\sum_{k\in I_t} G_{k,t}, \quad \sigma_t = \left(\frac{1}{n_t}\sum_{k\in I_t}(G_{k,t} - \mu_t)^2\right)^{1/2}, \\ A_{k,t} &= \frac{G_{k,t} - \mu_t}{\sigma_t + \epsilon}, \qquad \epsilon = 10^{-8}. \end{aligned} \tag{22}$$

The standard deviation uses the population denominator. For numerical stability, the implementation divides returns at each position by their maximum absolute value $m_t = \max_{k\in I_t} |G_{k,t}|$ and scales epsilon to ϵ/m_t. When $m_t > 0$, this gives Equation 22 in exact arithmetic. All-zero returns, zero standard deviation, and singleton positions yield zero advantage. Short trajectories supply no padding values. Alignment follows action index, so a STOP may be compared with an addition at the same position in another trajectory. Advantages are detached before policy optimization.

Hierarchical KL and entropy. The reference policy is a deep copy made before the first RL update, with all parameters frozen and evaluation mode enabled throughout training. At each visited state, both policies use the same legal action support. For $X \in \{\mathcal{K}, \mathcal{H}\}$ denoting KL or entropy, respectively,

$$X(s) = X_r(s) + \sum_{r\neq\text{STOP}} \pi_\theta(r \mid s)\left[X_z(s,r) + \sum_z \pi_\theta(z \mid r, s) \sum_{j\in\mathcal{V}(s,r,z)} X_{e_j}(s,r,z)\right]. \tag{23}$$

For $\mathcal{K}$, the terms are categorical or Bernoulli divergences in the current-to-reference direction; for $\mathcal{H}$, they are current-policy entropies. Conditional terms are weighted by current-policy probabilities. The sums cover all legal branches at the visited state, including when the sampled action is STOP. Edge terms are summed over legal predecessors, and logarithms are natural.

Sampled action log-probabilities retain their computation graphs. For a non-STOP action, KL and entropy reuse the stored sampling output and hence the same dropout realization. STOP decoding skips conditional heads; to obtain the full hierarchical distribution, the implementation restores the pre-sampling RNG state and repeats the forward pass. It checks that the reproduced role logits match the sampled logits elementwise. Thus the current-policy regularizers share the sampling dropout realization, while the reference policy always has dropout disabled.

With $M = \sum_k L_k$, including every final STOP position, the action-averaged loss is

$$\mathcal{L} = \frac{1}{M} \sum_{k} \sum_{t=0}^{L_k - 1} \left[-\operatorname{sg}(A_{k,t}) \log \pi_\theta(a_{k,t} \mid s_{k,t}) + \beta_{\mathrm{KL}} \mathcal{K}(s_{k,t}) - \beta_H \mathcal{H}(s_{k,t}) \right]. \tag{24}$$

Each recorded action position, including a probability-one STOP at the horizon, receives equal weight. KL and entropy enter the loss separately from the rewards and can contribute gradients at zero-advantage positions. The optimizer is Adam with fixed learning rate 10^{-4} and otherwise default parameters, without gradient clipping or a learning-rate schedule; $\beta_{\mathrm{KL}} = 0.01$, and $\beta_H = 0.001$. Only the construction policy parameters listed above are updated.

A.4 Training and Inference Procedures

The procedures below specify the current undiscounted setting. Ledger lookups reuse existing evaluations; each newly executed graph–query pair incurs execution cost, including group members, tool-related calls when applicable, and final summarization. Probe execution is confined to training.

Procedure A1: One training update.

1. **Inputs:** training query q, its scorer, fixed library, current policy π_θ, frozen RL-initial reference π_{ref}, training query pool, and evaluation ledger. Select a shared probe set P excluding q and encode the public task.
2. **Rollouts:** for each of K trajectories, initialize $\mathcal{G} \leftarrow \mathcal{G}_0$ and $\Phi \leftarrow 0$. Until termination, encode the current graph and sample a masked role-or-STOP action. Record the visited state and action log-probability with its computation graph. Retain sampling outputs, or replay pre-sampling RNG for STOP, to recover the full policy distribution with the same dropout.
3. **Addition:** if a role is selected, sample its realization and legal predecessor indicators, completing the recorded action probability. Append the unit to obtain $\mathcal{G}'$. Retrieve or execute parent and child probe evaluations; compute ΔU_P, ΔC, and ΔD. Record reward $d = \alpha \Delta U_P - \beta \Delta C - \gamma_R \Delta D$ and set $(\mathcal{G}, \Phi) \leftarrow (\mathcal{G}', \Phi + d)$. Return to the role decision; at the outer-unit horizon or when the configured budget permits no addition, mask all non-STOP roles and record the resulting probability-one STOP.
4. **Termination:** on STOP, score the final graph on q and store `terminal_reward` $= S(\mathcal{G}, q)$ and `terminal_shaping_reward` $= -\Phi$. Keep this STOP in the action sequence. Set its return to the sum of the terminal fields; leave previously stored addition rewards unchanged.
5. **Update:** compute returns backward for every trajectory, normalize by actual action position using Equation 22, and detach the advantages. Compute hierarchical KL and entropy on the recorded states using the retained or replayed current-policy outputs and the frozen reference in evaluation mode. Include the STOP positions in normalization and action averaging. Minimize Equation 24 with Adam. **Outputs:** updated construction-policy parameters and ledger.

Procedure A2: Inference.

1. **Inputs:** public query q, fixed library, trained policy, and frozen execution LLM. Disable policy dropout and initialize the empty graph.

2. Re-encode the graph and sample the role-or-STOP decision. On a role decision, sample its realization and legal predecessor indicators, append the unit, and repeat. At the outer-unit horizon or when the configured budget permits no addition, only STOP remains legal. Terminate when STOP is selected.
3. Execute the completed agent graph and summarize the expanded graph's sink results together with the public task and output requirements, or use the direct-answer fallback for an empty graph. **Output:** final answer.

B Proof of Return Preservation

This section proves Proposition 1 in Section 4.4, following potential-based reward shaping (Ng et al., 1999), and derives the return-to-go used in Appendix A. The construction episode includes its final STOP action, also when the outer-unit limit leaves STOP as the only legal choice.

B.1 Episode and Potential Conventions

Fix a query and a probe set shared throughout its trajectory group. Let an episode contain T action transitions, indexed by $t = 0, \ldots, T-1$, with STOP at index $T-1$. State s_t is the state before action t, and s_T is the terminal state after STOP. For N additions, $T = N + 1$; immediate STOP gives $T = 1$. The horizon limits additions while retaining the final STOP position.

During construction, the potential is defined relative to the initial graph using the fixed probes, structural complexity, and role repetition specified in Appendix A. Evaluation records and the accumulated potential can be included in the training state to make this quantity well-defined along the realized episode. The policy continues to observe only the public task and graph features. We set

$$\Phi_0 = 0, \qquad \Phi_T = 0. \tag{25}$$

Here Φ_{T-1} is the completed graph's pre-STOP potential. Settlement sets the terminal potential to zero and retains task score S. Define base rewards:

$$r_t^{\text{task}} = \begin{cases} 0, & 0 \le t < T-1, \\ S, & t = T-1. \end{cases} \tag{26}$$

Using the same discount η for shaping and return computation, each shaped reward is

$$r'_t = r_t^{\text{task}} + \eta\Phi_{t+1} - \Phi_t. \tag{27}$$

In particular, the STOP reward is $S - \Phi_{T-1}$. The implementation stores the score and settlement separately and combines them at the STOP position when computing returns.

B.2 Proof of Proposition 1

Proposition 1 (Return preservation). For a finite episode with the endpoint conditions in Equation 25, the shaped and base discounted returns coincide:

$$\sum_{t=0}^{T-1} \eta^t r'_t = \sum_{t=0}^{T-1} \eta^t r_t^{\text{task}}. \tag{28}$$

Proof. Substitute Equation 27 and collect the potential terms:

$$\begin{aligned}\sum_{t=0}^{T-1} \eta^t r'_t &= \sum_{t=0}^{T-1} \eta^t r_t^{\text{task}} + \sum_{t=0}^{T-1} \eta^{t+1}\Phi_{t+1} - \sum_{t=0}^{T-1} \eta^t \Phi_t \\ &= \sum_{t=0}^{T-1} \eta^t r_t^{\text{task}} - \Phi_0 + \eta^T \Phi_T \\ &= \sum_{t=0}^{T-1} \eta^t r_t^{\text{task}}.\end{aligned} \tag{29}$$

Every interior potential cancels with its counterpart from the adjacent transition, and both endpoint terms vanish. The identity holds for each realized episode, including immediate STOP and the probability-one STOP at the unit horizon. Taking expectations over task queries, probe selection, policy samples, and execution outcomes therefore gives the same expected return for every policy under the two reward definitions. Their maximizing policy sets coincide, including within the policy class used by MAGIC. □

For a general discount, the common return is $\eta^{T-1}S$. In the implemented setting $\eta = 1$, it is exactly S, so potential shaping preserves the expected terminal task-score objective in the main text. Utility, structural complexity, and role repetition determine the intermediate allocation of reward, with their net contribution settled at STOP.

B.3 Return-to-Go and Position-Aligned Credit

Applying the same cancellation from an arbitrary action position t gives

$$\begin{aligned}G'_t &= \sum_{j=t}^{T-1} \eta^{j-t} r'_j \\ &= G_t^{\text{task}} - \Phi_t + \eta^{T-t}\Phi_T \\ &= \eta^{T-1-t} S - \Phi_t.\end{aligned} \tag{30}$$

Thus, for the current undiscounted configuration,

$$G'_t = S - \Phi_t. \tag{31}$$

The first action has return S because its potential is zero. At later positions, the potential provides a state-dependent reference for the partial organization built before the action. For two trajectories with the same terminal score, $G'_{i,t} - G'_{j,t} = -(\Phi_{i,t} - \Phi_{j,t})$: their returns differ according to their pre-action graph potentials. Position-aligned normalization converts these state-relative returns into advantages that weight the sampled construction actions. At the final STOP position, Equation 31 gives precisely the score plus terminal settlement, including for a horizon-forced STOP.

Return preservation concerns the environment objective. The learning procedure subsequently normalizes returns within each action position and applies hierarchical KL and entropy regularization as specified in Appendix A. Keeping these operations separate makes explicit how shaping preserves the task objective while supplying the returns used by the implemented optimizer.

C Experimental Configuration and Protocols

This section separates shared resources from experiment-specific data allocations and run settings. Appendix A defines the method and algorithms; Appendix D contains the diagnostic protocols. Source split names describe dataset provenance, whereas training, probe, development, and evaluation describe an example's role in a particular experiment.

Experiment	Datasets	Protocol
D1	MMLU-Pro; TAT-QA	Appendix D.1
D2	Analytical configuration coverage	Appendix D
D3	MMLU-Pro; GSM8K; TAT-QA	Appendix D.3
Q1 / Q4	Eight main benchmarks / four efficiency benchmarks	Appendix C.3
Q2	MMLU-Pro; TAT-QA	Appendix C.4
Q3	MMLU-Pro; TAT-QA	Appendix C.5

Table 3: Experiment-to-protocol index. Data allocations and settings apply only to the experiments explicitly identified in each protocol.

C.1 Experiment Overview

C.2 Shared Resources and Evaluation Conventions

Benchmarks and data roles. The main evaluation covers eight benchmarks across four task families. The source inventory below describes the dataset collections, not the samples used by every experiment. Q1 allocations are listed in Appendix C.3; Q2, Q3, D1, and D3 have separate protocols. Scores are expressed as percentages. TAT-QA uses official F1, coding tasks use pass@1, and the remaining main tasks use accuracy. D1's TAT-QA diagnostic uses its explicitly specified strict-answer scoring. Dataset versions and membership lists must identify each run's snapshots and examples.

Table 4: Source dataset inventory for the main-table runs.

Benchmark	Source train	Source validation/development	Source test
MMLU-Pro	None	70	12,032
StrategyQA	2,061 under the official-code split of 2,290 public training questions	229 under that split	490 hidden examples
AQuA-RAT	97,467	254	254
GSM8K	7,473	None	1,319
HumanEval	None	None	164
LiveCodeBench	No standard training split	No standard development split	1,055 cumulative examples in `release_v6`
TAT-QA	13,215	1,668	1,669
TabFact	92,283	12,792	12,779

Answer extraction and scoring. The scoring metric for each main-evaluation benchmark follows Table 5. For training, rewards use the corresponding score on a scale from zero to one; TAT-QA uses official F1, with EM recorded separately in the documented reward configuration. Coding rewards are single-sample test-success indicators.

The D1 diagnostic uses a separate strict-answer protocol for TAT-QA. Its accuracy measurements and the official F1 scores in the main evaluation and component ablations are reported under their respective protocols. Diagnostic sampling and scoring details belong to Appendix D.

All main-evaluation methods share dataset-specific answer parsers and scorers. For classification and question-answering tasks, the parser extracts the final prediction and maps it to the benchmark answer space before comparison with the reference. Dataset-specific normalization handles number formatting, option markers, and short textual aliases where supported by the evaluation protocol. MMLU-Pro, AQuA, StrategyQA, TabFact, and GSM8K use accuracy; TAT-QA uses official F1. Missing or unparseable final predictions receive zero credit.

For MAGIC, the executor returns a JSON object containing `analysis` and `answer`. The `answer` field is passed to the shared scoring pipeline; the analysis text is not used to recover a missing

answer. The executor adds the internal `FINAL:` wrapper required by the scoring interface without an additional model call. Dataset answer contracts for MAGIC are specified in Appendix C.7.

Validation, recovery, and failure handling. The `analysis`/`answer` object permits additional fields while requiring the mandatory fields and their correct types. Invalid JSON, duplicate keys, missing required fields, and type errors are recorded as format failures. TAT-QA's nested `values`/`scale` object remains strictly validated.

An intermediate node that fails semantic or contextual validation may receive one repair call. Format errors do not uniformly trigger repair. When an intermediate response is truncated at the 2,048-token output limit, one additional attempt may request a concise, complete answer. Direct-answer and outer-finalizer nodes do not receive these format-repair or truncation-recovery calls.

Network exceptions, timeouts, HTTP 429 responses, and HTTP 5xx responses allow at most four total attempts, with backoff intervals of 1, 2, and 4 seconds. SDK-level retries are disabled.

Node failures are recorded individually. Other graph nodes and the outer finalizer may still produce a valid answer, which is scored normally. An internal failure event therefore does not by itself set the example's score to zero. If no valid final answer is obtained, the score is zero; failed examples are retained in the evaluation denominator.

Code evaluation and pass@1. HumanEval and LiveCodeBench-v6 use a custom evaluation entry point that executes Python programs in a Bubblewrap sandbox inside a local Lima Linux virtual machine. For HumanEval, the evaluator retains necessary imports and helper definitions from the public prompt, removes the placeholder definition of the target function, loads the final program, and executes the supplied test code through `check(candidate)` using the task's `entry_point`. Success requires all assertions to pass. For LiveCodeBench-v6, evaluation combines public and private test cases and supports both standard-input/output and function-call modes. Standard output is compared after stripping leading and trailing whitespace from each line, with numerical lines compared exactly using `Decimal`; function return values are compared using Python value equality. All test cases must pass.

Candidate execution is limited to 2 seconds of CPU time and 3 seconds of wall-clock time, with sandbox startup measured separately. The infrastructure has an overall 30-second wall-clock limit. Memory and output limits are 256 MiB and 64 KiB, respectively. For LiveCodeBench-v6, these execution limits are set when each test case starts, and CPU and execution wall-clock times are also accumulated across all cases and checked against the same 2-second and 3-second totals.

Pass@1 is the mean binary success score of one final program from one complete system run per question. Internal candidates, tool calls, and public-test executions are part of that run; hidden-test outcomes are used only to score the final submission. Syntax errors, runtime errors, failed tests, execution timeouts, and unparseable final answers receive zero credit. Sandbox unavailability and infrastructure timeouts also receive zero credit, with distinct execution statuses retained to identify their causes.

C.3 Overall Performance and Inference Efficiency (Q1 / Q4)

Q1 data allocation. Table 5 applies to the overall-performance comparison in Table 1. It does not specify the data pools for the component, optimization, or diagnostic experiments.

Source provenance and held-out evaluation. Source split names and experimental roles are distinct. In Q1, each benchmark has a designated training subset and a disjoint evaluation subset. For MMLU-Pro, these are separate subsets of the source test split; for StrategyQA, they are separate subsets of the pooled source development and training splits. These two evaluations therefore use custom held-out partitions, rather than the original benchmark evaluation partitions. A shared source split does not imply shared training and evaluation examples. HumanEval uses LiveCodeBench's 100 training examples and its own 164 evaluation examples. The reference implementation rejects overlapping training and evaluation task IDs. This check establishes ID-level separation, not semantic deduplication or document-level separation.

Table 5: Q1 training and held-out evaluation allocations. Source split names identify provenance; the two study subsets are disjoint.

Benchmark	Metric	Study training examples	Held-out evaluation examples
MMLU-Pro	Accuracy	100 from source test	500 from source test
StrategyQA	Accuracy	100 from the pooled source dev + train partitions	500 from the pooled source dev + train partitions
AQuA-RAT	Accuracy	100 from source train	All 254 source test examples
GSM8K	Accuracy	100 from source train	All 1,319 source test examples
HumanEval	pass@1	The same 100 LiveCodeBench training examples	All 164 source test examples
LiveCodeBench-v6	pass@1	100 sampled from the remainder of `release_v6`, excluding the v6 test subset	All 175 examples with `version_tag="v6"`
TAT-QA	Official F1	100 from source train	All 1,663 examples in the official test-gold release
TabFact	Accuracy	100 from source train	500 from source test

The main evaluation uses no development set for hyperparameter tuning or checkpoint selection. Each run evaluates the final checkpoint after the prescribed 100 updates. The allocation seed is fixed at 42 across runs; seeds 41, 42, and 43 vary training randomness rather than dataset membership.

Main-table sampling. For MMLU-Pro, 600 indices are drawn without replacement from the source test split using Python `random.Random(42)`. The sampled indices are then sorted by source index. The first 500 selected indices are assigned to evaluation and the last 100 to training.

StrategyQA is formed by concatenating the development and training partitions in `dev + train` order and sampling 600 examples with seed 42. The sampled examples are sorted by their indices in the concatenated pool; the first 500 are assigned to evaluation and the last 100 to training.

AQuA-RAT, GSM8K, and TAT-QA each use 100 training examples selected with seed 42. AQuA-RAT and GSM8K use their complete source test splits. For TAT-QA, we evaluate all 1,663 questions in the official test-gold release and report official F1 over these questions. The original test release contains 1,669 questions; the publicly released gold file contains 1,663 questions with reference answers. LiveCodeBench evaluation uses all 175 examples with `version_tag="v6"` in the cumulative `release_v6` collection. After excluding this test subset, 100 examples are sampled from the remaining collection with seed 42 for training. HumanEval reuses these same 100 training examples and evaluates on all 164 HumanEval test examples. TabFact uses 100 source training examples and 500 source test examples, selected with seed 42. MAGIC uses random initialization with seeds 41, 42, and 43 for the three Q1 runs and no behavior-cloning initialization.

For Q1, each dataset-specific policy is trained independently in three runs with seeds 41, 42, and 43. In each run, it is trained for one pass over its 100 training examples. Each example yields eight trajectories ($K = 8$) and one optimizer update, giving 800 trajectories and 100 updates per run. The policy sampling temperature decreases linearly from 1.2 to 1.0 over these updates, and evaluation uses the final policy weights. Q4 reuses these trained weights and the same training and evaluation subsets as Q1. The dataset allocations described above use sampling without replacement with seed 42. Unless an explicit release is specified above, datasets use the latest versions available when the experiments were conducted.

Dataset-specific training and utility probes. A separate construction policy is trained for each dataset. For each training query, four utility probes are selected without replacement from the other 99 examples in the designated training subset using a reproducible, seed-dependent pseudorandom hash ordering. They remain fixed throughout that trajectory group. Probes are training examples, not an additional validation or evaluation set; their reference answers provide training supervision through reward computation. Neither the current query nor any held-out evaluation example is eligible for probe selection. Reference answers and code tests are excluded from policy observations.

During final evaluation, reference answers and hidden code tests are used for scoring the final outputs, without policy updates or training-time probe evaluation.

Baseline evaluation. We compare MAGIC with 17 baselines across the eight benchmarks. All methods use a shared dataset-specific scoring pipeline for answer extraction, normalization, and metric calculation. The baselines are grouped into the following three categories.

Table 6: Baseline categories in the main evaluation.

Category	Methods
Direct / single agent	DeepSeek V4 Flash; Qwen Plus
Established MAS	AutoGen; LLM-Debate
Adaptive / learned organization	DyLAN; GPTSwarm; G-Designer; Sparse-Comm; GraphSearch; Graph-R1; R-GFM; BIGMAS; MasRouter; GoAgent; VeriMAP; ARG-Designer; EIB-Learner

The direct single-agent baselines use `deepseek-v4-flash` and `qwen-plus`, respectively. MAGIC uses the initial release of `deepseek-v4-flash` as its Q1 executor.

Q4 result reuse and token accounting. For methods shared with Q1, the efficiency analysis reuses the Q1 mean performance results on MMLU-Pro, GSM8K, HumanEval, and TabFact and compares them with inference-token consumption. Figure 5 also includes Complete, Random, Tree, and Chain organizations, which are not part of Table 1. Rankings in Q4 refer to this displayed comparison set. For each method, performance and total inference tokens are averaged over the three runs with seeds 41, 42, and 43. Each run's token total sums input and output usage over its complete evaluation set; the plotted value is the mean of these three totals, not their sum or a per-question mean. Performance and token measurements use the corresponding execution records. Q4 does not introduce a separate training objective or new dataset allocation. Inference performance–cost measurements count both input and output tokens across all LLM calls. Accounting includes failed attempts, retries, repair calls, and every LLM turn before and after tool execution, as well as group-internal nodes and the outer finalizer. Internal recovery costs are therefore included alongside successful calls. Training-resource accounting is kept separate from final inference consumption.

C.4 Component Ablations (Q2)

Expanded-node budget and graph limits. All five variants use $N_A + 4N_G \leq 8$, where N_A and N_G count outer atomic and group units. Each group contains three workers and one internal aggregator. The outer summarizer is excluded from this node budget and included in reported execution costs. The maximum outer-unit count and expanded depth are both eight. A DAG with at most eight expanded nodes cannot exceed depth eight, so the depth limit does not further restrict budget-feasible connections. All variants use the same 8×256 position and degree embedding capacities. The four trained variants are trained under these limits; Initialization only is evaluated before updates. These settings are specific to Q2 and do not replace the Q1/Q3 limits.

An addition must fit within the remaining expanded-node budget: an atomic realization requires one node, and a group requires four. Realizations that exceed the remaining budget are masked, and a role with no admissible realization is masked at the role level. When no addition remains feasible, only STOP is legal; early STOP remains available throughout. The budget constrains execution nodes, not all API calls, since tool use, recovery, and final summarization can add calls.

Component comparisons. On MMLU-Pro and TAT-QA, All-Atomic and All-Group restrict realizations during both training and inference. Initialization only evaluates the policy before optimization. Terminal-only RL retains the mixed action space and RL training while removing dense utility and structural rewards. These four variants are compared with Full using DeepSeek V4 Flash on the same evaluation examples within this experiment. MMLU-Pro uses accuracy and TAT-QA uses official F1. These allocations are specific to Q2 and are not inherited from the Q1 data table.

Data partition and training. For MMLU-Pro, we read the locally cached official test split in its original file order and shuffle the rows using Python's `random.Random(23)`. The first 100 questions are used for evaluation and the next 100 for policy training. These are disjoint subsets constructed within the official test split, without subject stratification or filtering by difficulty or model performance. For TAT-QA, we partition the locally cached official development split by table UID, defined as the prefix before the colon in each question's `uid`. Groups are listed in first-occurrence order and shuffled using `random.Random(23)`. Preserving the original question order within each group, we select 100 evaluation questions followed by 100 training questions from subsequent, disjoint groups. Unused questions in boundary groups are discarded. This separates tables and their associated context, rather than entire annual reports; no stratification by question type, difficulty, or answer category is applied. All variants share the same partition within each dataset. Each trained variant undergoes 100 updates with two trajectories per query; Initialization only receives no updates.

For each dataset, Full, Terminal-only RL, All-Atomic, and All-Group each make one pass over the 100 training queries. Each query yields a group of two trajectories and one optimizer update, giving 200 trajectories and 100 updates per configuration. Training starts from random initialization without STOP warmup. Evaluation uses the final parameters after update 100 in evaluation mode, with no intermediate-checkpoint selection or early stopping. Initialization only evaluates the randomly initialized parameters with zero updates. Training-query, trajectory, and update counts are matched; tokens and API costs vary with granularity, probes, and ledger reuse.

Q2 uses initialization seed 23, with all configurations on a given dataset sharing the same initial policy weights. During the 100 training updates, the policy sampling temperature decreases linearly from 1.2 to 1.0. Inference uses probabilistic sampling at temperature 1.0 with dropout disabled. Apart from the specified ablations, all Q2 variants share the same expanded-node budget, graph limits, and embedding capacities described above. The core architecture, optimizer, regularization, and executor settings follow Table 7. Dense-reward configurations use four fixed probes per trajectory group and the listed shaping weights, discount, and free node/edge allowances. Training-query, trajectory, and update counts are controlled; realized training API expenditure is not matched.

Execution and reporting. DeepSeek V4 Flash uses JSON-object output, temperature zero, thinking disabled, and a 2,048-token output limit. Evaluation samples graphs at policy temperature one with dropout disabled. Full and the fixed-granularity variants use the potential-based reward; Terminal-only RL sets the three shaping weights to zero. Table 2 reports mean calls per evaluation question and mean input-plus-output tokens per question in thousands. Token totals are divided by 100 questions and by 1,000; call totals are divided by 100. Accounting includes group workers, internal aggregators, the outer summarizer or empty-graph direct-answer path, post-tool responses, and recovery attempts, excluding training calls. Failed examples remain in the denominator. A/G reports the corresponding mean counts of outer atomic and group units; internal group members and the outer summarizer are not counted as outer atomic units.

Across the five variants, mean calls range from 8.10 to 8.40 on MMLU-Pro and from 8.80 to 9.10 on TAT-QA. The maximum-to-minimum relative spreads are 3.70% and 3.41%, respectively. These are comparable mean call counts, not identical token usage or per-query costs. Full averages 2.20 atomic units and 1.20 groups on each dataset, equivalent to seven expanded nodes on average. Aggregate A/G counts do not determine the frequency of mixed graphs, their edge counts, or their connection patterns. Q2 uses different evaluation subsets and graph limits from Q1 and Q3, so absolute scores and token totals across these experiments should not be interpreted as paired comparisons.

C.5 Optimization Strategy Comparison (Q3)

Data and sample selection. Evaluation uses 200 MMLU-Pro source-test queries and 200 TAT-QA source-dev queries. For each dataset, the original training experiment's 100 training examples and 100 evaluation examples are excluded; TAT-QA exclusions apply to the corresponding complete table UIDs. Remaining IDs are sorted, shuffled with seed 23, and the first 200 are selected. The selected memberships are recorded in the dataset-specific `membership.json` files.

Comparison and training budget. The comparison includes Search-then-SFT, Terminal-only RL, and Dense-reward RL. Terminal-only RL uses final task scores without potential-based shaping, using the same reward ablation as Q2 but retaining Q3's own graph limits and evaluation set. The

detailed run accounting below covers Search-then-SFT and Dense-reward RL. Both routes cycle through a fixed set of 100 training questions per dataset and start from identical random weights with seed 23. They use a one-layer, 256-dimensional edge-aware GRU with training dropout 0.1, at most three outer units and expanded depth four, and Adam with learning rate 10^{-4}. Both start from the same ledger snapshot and subsequently maintain separate caches; cached graph–question scores incur no additional API cost. The comparison controls DeepSeek V4 Flash API expenditure, not parameter-update counts. Checkpoints are saved at initialization and CNY 2, 4, 6, 8, and 10 budget stages. Costs use call usage and official pricing, with complete-batch accounting allowing small budget overshoots; evaluation costs are separate.

Search and demonstration selection. For each question, Search-then-SFT generates eight distinct candidates, including the empty graph, using seeded random roles and granularities over single-node, two-node serial/parallel, three-node converging, and random legal DAG shapes. Candidates are generated independently of the policy network. Scored candidates accumulate in a per-question pool. If the best score is positive, one highest-scoring graph is subjected to legal single-node and single-edge deletions, which are scored and added to the pool. Up to four highest-scoring positive candidates are retained, breaking ties by graph hash. Thus MMLU-Pro demonstrations are correct answers, whereas TAT-QA demonstrations have the highest positive F1 in the pool, not necessarily F1 one.

Stage-wise SFT and RL updates. At each nonzero budget stage, SFT runs one epoch over questions with a demonstration, selecting one graph per question and rotating tied demonstrations across stages. Each graph is serialized into node-addition actions followed by STOP. One Adam update per question minimizes role and granularity cross-entropy plus legal-predecessor binary cross-entropy, averaged over action positions including STOP. Demonstrations may be reused across stages. SFT incurs no LLM API calls and uses no RL KL or entropy regularizers; search resumes after each stage. Dense-reward RL samples two current-policy trajectories per question and updates once per group. It uses the final potential-based reward with four shared fixed probes, discount one, shaping weights $(1, 0.02, 0.05)$, node/edge free allowances of six, and zero terminal call/token penalties. Its KL and entropy coefficients are 0.01 and 0.001.

On MMLU-Pro, Search-then-SFT performs 411 search batches and 423 SFT updates at CNY 10.0121, while Dense-reward RL performs 215 updates at CNY 10.0748. On TAT-QA, the corresponding totals are 488 search batches and 458 SFT updates at CNY 10.0104, versus 194 RL updates at CNY 10.0497.

Evaluation settings. Execution uses DeepSeek V4 Flash with `json_object`, thinking disabled, temperature zero, and a 2,048-token output limit. Graph construction is probabilistic with dropout disabled; seed 23 and the query ID determine the construction randomness. The API concurrency limit is eight. The common initial policy is evaluated once per dataset and its results are shown for both original routes. Across their stages, evaluation comprises 4,400 actual MAS executions and 4,800 reported rows; utility-ledger answers are not reused. These counts cover Search-then-SFT and Dense-reward RL only and exclude the Terminal-only RL evaluations shown in the three-way comparison.

Scoring and accounting. MMLU-Pro uses accuracy; TAT-QA uses official F1, with EM recorded separately. Valid final-answer scores and execution-failure events are recorded separately. Evaluation-token records include inputs, outputs, post-tool turns, summarization, and retries with returned usage. Shared initialization is displayed under both routes but counted only once in physical execution totals. These evaluation tokens are distinct from the training API cost on the budget axis. Results are aggregated over the 200 evaluation examples per dataset at each saved stage.

C.6 Implementation and Reproducibility

Reference implementation settings and run-specific overrides. Table 7 summarizes the shared architecture and experiment-specific graph limits. Q1 and Q2 Full share the graph-encoding and action-head design, optimizer and learning rate, regularization, reward settings, probe count, and executor decoding settings. Q2 changes the graph limits and embedding capacity and adds an expanded-node budget. Q1 uses initialization seeds 41, 42, and 43, while Q2 uses seed 23; both use DeepSeek V4 Flash. Q4 reuses the Q1 policies and configuration for overlapping methods. The Q2

ablations modify the components specified above. Q3 uses DeepSeek V4 Flash under its separate protocol. D1's expanded-depth limit is six and D3's diagnostic group size is eight.

Table 7: Reference implementation settings.

Component	Setting
Frozen semantic encoder	`sentence-transformers/all-MiniLM-L6-v2`, 384 dimensions
Graph encoder	One shared edge-aware GRUCell, hidden size 256
GRU input dropout	0.1 during training; disabled at inference
Maximum outer units / expanded depth	Q1/Q3: 3 / 4; Q2: 8 / 8. Outer summarizer excluded from depth
Expanded execution-node budget	Q2: $N_A + 4N_G \leq 8$, excluding the outer summarizer
Position / degree embedding capacity	Q1/Q3: 3×256 per table; Q2: 8×256 per table
Q1/Q2 training policy temperature	Linear annealing from 1.2 to 1.0 over 100 updates
Inference policy temperature	1.0; probabilistic sampling
Initialization	Random; seeds 41, 42, 43 for Q1; seed 23 for Q2; no behavior-cloning initialization or STOP warmup
RL trajectories per query group	Q1: 8; Q2/Q3: 2; D3: 8
Fixed utility probes per group	4 other training queries
Optimizer / learning rate	Adam with default parameters / fixed 10^{-4}
KL / entropy coefficients	0.01 / 0.001
Advantage epsilon	10^{-8}, scaled consistently during numerical normalization
Reward weights $(\alpha, \beta, \gamma_R)$	$(1, 0.02, 0.05)$
Shaping and return discount	1
Utility smoothing (κ, u_0)	$(1, 0.5)$
Expanded node / edge weights	1 / 1
Free expanded node / edge allowances	6 / 6
Terminal call / token penalty weights	0 / 0
Executor temperature / thinking	0 / disabled
Executor output mode / output limit	`json_object` / 2,048 tokens

For Q1, each of the three runs includes the complete training and evaluation procedure, with one complete system execution per evaluation question. We first aggregate question-level scores within each run, then report the mean and standard deviation of the three benchmark scores. Q2 reports one run per variant and dataset, aggregating its 100 evaluation questions. Q4 reports three-run means for both performance and evaluation-set token totals, reusing Q1 performance scores for overlapping methods; its inference-token accounting is specified in Appendix C.3.

Adam uses default parameters apart from the learning rate, with no gradient clipping or learning-rate schedule. Policy sampling and executor decoding use separate temperatures. The unit horizon and depth bound constrain organization structure; the free node and edge allowances define the potential. The reward configuration's call and token normalizers do not impose execution hard limits.

The reference policy is frozen before the first RL update and remains in evaluation mode. The current policy's regularizers reuse the sampling dropout realization, including RNG replay for STOP, as detailed in Appendix A. Reproducibility therefore depends on policy sampling and dropout randomness as well as the dataset and probe selection procedures.

Run count and aggregation. Q1 reports the mean and standard deviation over three complete runs with seeds 41, 42, and 43 for each method and benchmark. For trainable methods, each run includes training and evaluation; direct inference methods repeat evaluation under the corresponding run seeds. Q2 and Q3 use one run per dataset and experimental condition, aggregating evaluation examples within that run. Q3's budget stages are checkpoints within each route's run, not independent repeats.

Q1 uses seeds 41, 42, and 43 and Q2 uses seed 23 for policy initialization, policy sampling, and training dropout. D3 uses eight trajectories per query under its separate diagnostic protocol.

Software, hardware, and resource measurement.

C.7 Role Libraries, Prompts, and Execution Interfaces

Role library and realizations. The documented framework registers 14 top-level roles, each with an atomic and a group realization. A dataset profile selects the available roles and tools; the construction policy selects the units that actually execute. An atomic realization uses one executor. Each group contains three workers feeding one internal aggregator, with no direct worker-to-worker edges. Table 8 lists the registered roles and their internal assignments. These library definitions remain fixed during policy training and inference.

Table 8: Registered roles and group realizations.

Top-level role	Atomic responsibility	Three group workers	Internal aggregator
`task_decomposer`	Identify constraints, subproblems, dependencies, and output requirements	`constraint_extractor`; `subproblem_planner`; `dependency_checker`	`decomposition_aggregator`
`evidence_analyst`	Extract, rank, and reconcile public evidence	`evidence_finder`; `relevance_ranker`; `contradiction_finder`	`evidence_aggregator`
`independent_solver`	Solve independently and assess predecessor conclusions	`solver_a`; `solver_b`; `assumption_checker`	`solution_consensus`
`adversarial_verifier`	Check counterexamples, consistency, omissions, and answer format	`counterexample_searcher`; `consistency_checker`; `format_checker`	`verification_aggregator`
`knowledge_reasoner`	Apply model knowledge and identify uncertainty	`fact_recaller`; `contextual_reasoner`; `uncertainty_calibrator`	`knowledge_aggregator`
`option_eliminator`	Assess support and objections for candidate options	`option_support_analyst`; `option_refute_analyst`; `distractor_detector`	`option_aggregator`
`logic_reasoner`	Formalize premises and perform logical, causal, and Boolean reasoning	`premise_formalizer`; `forward_reasoner`; `countermodel_searcher`	`logic_aggregator`

Role responsibilities are specified through prompts. Intermediate roles may supply a complete candidate answer or `null` when unable to answer. Aggregators combine their workers' contributions, retaining relevant disagreements, uncertainty, and failed checks. Roles named "finder" or "searcher" examine public task information or model knowledge; the tool registry excludes web search.

Tool access. The executor supports four tools. `calculator` evaluates restricted expressions; `python_exec` runs a standalone Python program in a sandbox and returns stdout, stderr, and exit status; `public_code` performs public code checks and sample tests; and `table_reader` accesses the task's public table data. Public code tests use either standard-input/output comparisons or function-call comparisons with supplied expected outputs. Hidden benchmark tests remain part of final scoring.

Other registered execution roles have no tool permission. Actual availability is the intersection of executor-level permissions and tools enabled by the dataset profile, subject to runtime availability. Each group worker has its own permissions; atomic-role permissions are not automatically inherited by the group members or aggregator.

Content and tool-request protocol. Protocol `analysis-answer-v2` requires `analysis` and `answer`, allowing extra fields:

```
{
  "analysis": "Brief reasoning, evidence, checks, and uncertainty",
  "answer": "Complete candidate answer"
```

Table 9: Registered roles and group realizations. (continued).

Top-level role	Atomic responsibility	Three group workers	Internal aggregator
quantitative_modeler	Define variables, equations, units, and a solution plan	variable_mapper; equation_builder; unit_constraint_checker	model_aggregator
calculation_auditor	Recompute numerical results and identify discrepancies	primary_calculator; independent_recalculator; magnitude_unit_checker	calculation_aggregator
algorithm_designer	Develop algorithms and assess complexity and boundary cases	algorithm_candidate_a; algorithm_candidate_b; complexity_edgecase_analyst	algorithm_aggregator
code_implementer	Implement the required function or input/output interface	implementation_drafter; interface_specialist; public_sample_runner	code_aggregator
code_reviewer	Review correctness, complexity, and boundary behavior	static_analyzer; edge_case_generator; complexity_reviewer	review_aggregator
table_interpreter	Locate rows, columns, entities, time references, and scales	row_locator; column_relation_analyst; temporal_scale_checker	table_aggregator
entailment_judge	Assess a claim using supporting and refuting evidence	support_case_builder; refutation_case_builder; quantifier_scope_checker	entailment_aggregator

Table 10: Executor-level tool permissions.

Atomic role or group worker	Permitted tools
calculation_auditor, primary_calculator, independent_recalculator, magnitude_unit_checker	calculator, python_exec
code_implementer, public_sample_runner	public_code, python_exec
code_reviewer, static_analyzer, edge_case_generator	public_code
table_interpreter, row_locator, column_relation_analyst, temporal_scale_checker	table_reader

```
}
```

`analysis` must be a nonempty string. `answer` follows the dataset-specific contract in Table 11 and may be `null` when a node cannot answer. The model returns JSON without Markdown fences or a `FINAL:` prefix. Ordinary atomic nodes, group workers, and internal aggregators wrap their content in a node-response envelope:

```
{
  "kind": "content",
```

```
  "content": {
    "analysis": "Checked the public evidence",
    "answer": "42"
  }
}
```

A permitted tool is requested with the alternative envelope:

```
{
  "kind": "tool_request",
  "tool_request": {
    "tool": "python_exec",
    "payload": {
      "source": "print((120 - 100) / 100 * 100)"
    }
  }
}
```

Each node execution permits at most one tool request. After the executor supplies the actual tool result, the next model response must submit content. A normal execution therefore uses one model response without a tool or two with a tool; intermediate-node repair, truncation recovery, and transport retries may add API calls under the rules in Appendix C.2. The runtime supplies tool observations, status, timing, and token-accounting fields. These fields are not generated by the model.

Routing and finalization. Parsed outputs are wrapped in runtime packets carrying provenance, task association, and execution status. Downstream requests contain the public task, role responsibilities, answer contract, and projected messages from graph-specified predecessors. Message projection and length budgets determine the content delivered to each node. Gold answers and hidden tests are excluded from solving inputs.

All three group workers receive every external direct predecessor's result. The internal aggregator receives the three workers as its direct predecessors and emits the group's external result. The outer finalizer receives the public question, output contract, and packets from all sinks of the expanded graph. The finalizer and the documented direct single-LLM path emit the core `analysis`/`answer` object without the ordinary node's `kind` envelope. Appendix A details construction and execution.

Table 11: Required model answer formats.

Dataset	`answer` format
MMLU-Pro	Option-label string, e.g., `"A"`
AQuA-RAT	Option-label string, e.g., `"B"`
StrategyQA	String `"true"` or `"false"`
TabFact	String `"true"` or `"false"`
GSM8K	Numeric string without units, currency symbols, or commas; integers omit `.0`
HumanEval	Complete function source as a string without code fences
LiveCodeBench-v6	Complete code string matching the problem interface, without code fences
TAT-QA	Object with `values` and `scale` fields

Dataset answer contracts. StrategyQA and TabFact use string values rather than JSON Booleans. For TAT-QA, each answer span is a separate string in `values`; arithmetic and counting answers contain a single numeric value as required by the prompt. Scales are the empty string, `thousand`, `million`, `billion`, or `percent`. Numeric answers omit unit suffixes:

```
{
  "analysis": "Compute the growth rate from the relevant years",
  "answer": {
    "values": [
      "15.64"
```

```
    ],
    "scale": "percent"
  }
}
```

The runtime deterministically converts the validated final `answer` into the representation required by the scorer, including its internal `FINAL:` wrapper. The model itself does not emit this wrapper, and conversion requires no model call. Format validation and task scoring are separate: the scorer evaluates the answer's numerical, textual, and scale correctness after conversion.

Prompt and profile artifacts. These tables and examples specify the library and communication contracts. Exact role prompts and dataset profiles are separate artifacts.

D Extended Empirical Investigation

This section supplements the three diagnostics in the main-text Empirical Investigation: local granularity preferences, verified configuration coverage, and construction-level credit.

D.1 Controlled Granularity Assignments

Protocol. We fix a serial decomposer–solver–verifier skeleton and enumerate its eight atomic/group assignments: AAA, AAG, AGA, AGG, GAA, GAG, GGA, and GGG. The letters follow the role order; A denotes an atomic realization and G a group realization. All assignments use the same 200 MMLU-Pro test queries and 200 TAT-QA development queries, with one run per query and assignment. The role library, execution model, prompts, and decoding settings remain fixed across assignments. This gives 1,600 query–configuration executions per dataset.

For each dataset, the adapter produces a stable `example_id`. Examples are sorted by the ascending SHA-256 digest below, and the first 200 are selected:

```
key = hashlib.sha256(f"d1-v1\0{42}\0{example_id}".encode()).digest()
```

This seed-42 selection is unstratified. TAT-QA is sampled at the question level, rather than by table or document. All eight assignments share the selected examples. D1 sets the maximum expanded depth to six, allowing all eight serial assignments, including GGG. This diagnostic differs from Q1/Q3's depth-four setting and does not use Q2's eight-node budget. It reports accuracy and total input/output tokens on its diagnostic-specific sets.

Historical TAT-QA scoring. D1 uses a custom binary scorer that extracts an answer string from a unique terminal `FINAL:` marker. This historical protocol precedes the official F1 scorer and the current `values`/`scale` object contract. Its principal rules are:

Table 12: D1 TAT-QA binary scoring rules.

Answer type	Correctness rule
Arithmetic	Parse prediction and reference as Decimal values, round both to two decimal places using ties-to-even, and compare for equality.
Count	Compare Decimal values exactly, without two-decimal rounding.
Numeric syntax and scale	Accept thousands separators, leading currency symbols, and parentheses for negatives. A nonempty reference scale requires a matching `percent`, `thousand`, `million`, or `billion` suffix; `%` is also accepted for percent. No cross-scale conversion is applied.
Single text span	Apply Unicode NFKC and case normalization, collapse whitespace, strip designated boundary punctuation, and require exact equality.
Multiple spans	Split on semicolons, normalize each span, and compare multisets using `Counter`, ignoring order while retaining multiplicity.
Numeric span	Apply numeric normalization and two-decimal comparison; compare scale as well for scaled numeric spans.

The recorded correctness also requires execution without a failure event:

```
correct = score.correct and not result.failures
```

Consequently, a correct answer accompanied by a recorded MAS execution failure receives zero credit in D1. This diagnostic-specific rule is distinct from the final-answer scoring protocol in Appendix C, which records internal failures separately.

Role-conditioned comparisons. To isolate a local realization change, we compare a mixed assignment with AAA while keeping the other two roles atomic. The final-answer accuracy changes reported in the main text are:

Table 13: D1 role-level granularity interventions.

Intervention relative to AAA	MMLU-Pro change (percentage points)	TAT-QA change (percentage points)
Group only the decomposer: GAA	+4.5	0.0
Group only the verifier: AAG	−1.0	+2.5

These differences concern the diagnostic correctness measure after the full fixed skeleton executes. MMLU-Pro favors GAA, reaching 81.5% accuracy versus 77.0% for both fixed endpoints, whereas TAT-QA favors AAG, reaching 70.0% versus 67.5%. Relative to GGG, these mixed assignments reduce token consumption by 58.72% and 48.15%, respectively. On each dataset, AAA and the preferred mixed assignment form the observed accuracy–token frontier: the mixed assignment increases accuracy at additional cost over AAA, while using fewer tokens than GGG. The controlled assignments therefore identify task-dependent choices about where to allocate group execution.

D.2 Verified Configuration Coverage

Fix the role sequence and outer dependency skeleton, with m positions that each admit both atomic and group realizations. The diagnostic counts their binary realization assignments:

$$\mathcal{Z}_{\text{mix}} = \{A, G\}^m, \qquad |\mathcal{Z}_{\text{mix}}| = 2^m. \tag{32}$$

Let V_B be the set of distinct assignments verified under a budget of B complete graph executions. Since each verification consumes at least one such execution,

$$|V_B| \le \min(2^m, B), \qquad \text{Coverage}(B) = \frac{|V_B|}{2^m} \le \min\left(1, \frac{B}{2^m}\right). \tag{33}$$

Repeated executions of an already verified assignment consume budget without adding a distinct assignment to V_B. The denominator counts the binary assignments for the fixed skeleton; additional role orders and edge choices expand the construction space. This counting argument relates verified configuration coverage to graph-execution budget.

D.3 Construction Credit on Shared Trajectories

Data and diagnostic updates. The DeepSeek V4 Flash diagnostic uses 30 distinct queries per dataset from MMLU-Pro validation, GSM8K train, and TAT-QA test. Each dataset has an independently initialized policy trained with the model and reward defined in Section 4, rather than restored from an earlier diagnostic checkpoint. Each query is used for one update with eight trajectories, giving 30 updates and 240 trajectories per dataset, or 720 trajectories overall. Here, TAT-QA "test" denotes the source split: the selected questions are used for D3 diagnostic updates, not as a held-out test of the D3 policy. This experiment compares credit signals on shared training trajectories; its measurements should not be interpreted as held-out generalization scores. Each query group shares four fixed probes drawn from other training queries. The eight-trajectory group size is diagnostic-specific.

Experimental chronology and separation from Q1. The Q1 method and experimental configuration were fixed before D3 was conducted. D3 was performed solely as a subsequent analysis of construction credit signals. No D3 policy weights, cached execution results, or diagnostic feedback were used in Q1 training, evaluation, or method and configuration selection. In particular, the TAT-QA source-test questions used for D3 diagnostic updates did not supply training rewards or utility probes to Q1. The use of source-test data in this subsequent diagnostic therefore does not change the Q1 training and held-out evaluation allocation described in Appendix C.3.

Diagnostic return definitions. Terminal-only and dense returns are computed on the same recorded trajectories. Dense rewards follow Section 4.3 and Appendix A.3, including full potential settlement at both sampled and horizon-forced STOP. With discount $\eta = 1$, the two return-to-go definitions are

$$G_{i,t}^{\text{terminal}} = S_i, \qquad G_{i,t}^{\text{dense}} = S_i - \Phi_{i,t}. \tag{34}$$

Here S_i is the terminal task score and $\Phi_{i,t}$ is the potential before action t. Both initial returns equal S_i because $\Phi_{i,0} = 0$. The implementation checks that each complete shaped return equals its terminal score and that both initial returns agree.

Non-zero action advantages. Within each eight-trajectory query group, returns are normalized at each action position over the trajectories that contain that position:

$$A_{i,t} = \frac{G_{i,t} - \mu_t}{\sigma_t + 10^{-8}}, \qquad \text{non-zero iff } |A_{i,t}| > 10^{-8}. \tag{35}$$

The mean and standard deviation are computed separately for each return definition. Zero-standard-deviation positions receive zero advantage. Normalization includes all trajectories with an actual action at that position, including STOP; shorter trajectories contribute no padding. A STOP and an addition may therefore be compared at the same index. After normalization, the primary statistic selects only intermediate additions, $\mathcal{I} = \{(i,t) : t > 0,\ a_{i,t} \text{ is ADD}\}$, and reports $\rho = \sum_{(i,t)\in\mathcal{I}} \mathbf{1}[|A_{i,t}| > 10^{-8}]/|\mathcal{I}|$. Counts are also reported by construction position. Initial decisions, subsequent sampled STOP, and horizon-forced STOP are reported separately from this primary statistic. Empty reporting sets are marked N/A.

Equal-terminal-reward pairs. Within each update, we enumerate unordered pairs among that query's eight trajectories. We retain equal-terminal-score pairs satisfying

$$|S_i - S_j| < 10^{-8}. \tag{36}$$

For each retained pair, an eligible comparison position has $t > 0$ and an actual ADD action in both trajectories. A trajectory-pair–position is counted as distinguished when

$$|G_{i,t}^{\text{dense}} - G_{j,t}^{\text{dense}}| > 10^{-8}. \tag{37}$$

The denominator is the number of eligible trajectory-pair–positions, each counted once within its query group. One pair can contribute multiple positions. Returns are computed from the complete recorded reward sequence, including terminal settlement; shorter trajectories are not padded. The statistic is N/A when no eligible comparisons exist.

Pooled statistics. Table 14 reports pooled counts across updates for each dataset. Percentages are ratios of pooled numerators and denominators, rather than averages of per-query percentages.

Potential-based shaping supplies non-zero relative advantages to a larger fraction of intermediate ADD actions and distinguishes some equal-terminal-score trajectories at intermediate positions while preserving complete task returns as established in Proposition 1. The two action-statistic columns compare return definitions on the same trajectory sample, rather than independent training runs.

Table 14: D3 statistics with final potential-based rewards: 30 queries and 240 trajectories per dataset.

Dataset	Terminal-only non-zero advantages / intermediate ADD actions	Dense non-zero advantages / intermediate ADD actions	Distinguished / eligible equal-score pair–positions
MMLU-Pro	105/320 (32.81%)	266/320 (83.13%)	343/642 (53.43%)
GSM8K	5/141 (3.55%)	47/141 (33.33%)	48/204 (23.53%)
TAT-QA	39/236 (16.53%)	181/236 (76.69%)	123/376 (32.71%)