SMART: Scalable Multi-Agent Role-conditioned Teaming via LLM-free Tree Search

Runzhe Zhang[1], **Ning Li**[1], **Xingshan Zeng**[2], **Letian Chen**[1,4], **Yasheng Wang**[3], **Weinan Zhang**[1], **Yong Yu**[1], **Weiwen Liu**[1✉]

[1]Shanghai Jiao Tong University [2]Huawei Technologies [3]Tencent [4]Shanghai Innovation Institute

Abstract

Large language model (LLM)-powered multi-agent systems have been shown to effectively solve real-world end-to-end problems through collaboration among diverse agents. However, as LLM capabilities advance and user demands continue to grow, the population of available agents and tools continues to increase, giving rise to open and evolving ecosystems of autonomous agents. In this paradigm, agents are likely to become increasingly specialized, each focusing on particular capabilities or problem types. In such a setting, a central challenge is how to efficiently assemble an appropriate team of agents to take on different responsibilities from a massive agent pool for a given task. In this work, we study this problem in a controlled setting with 300 task-oriented agents. SMART (Scalable Multi-Agent Role-conditioned Teaming) first plans a task-conditioned collaboration topology and candidate role set, then constructs a compact role-conditioned candidate pool, and finally uses an LLM-free Monte Carlo Tree Search (MCTS), which makes no LLM call inside search or rollouts, to assign agents to roles online via metadata-based rollouts. This design enables high flexibility without incurring prohibitive inference costs. Across multiple experiments, our approach achieves superior performance on GSM8K (93.64%), MATH (61.73%), HumanEval (93.89%), and MBPP (87.68%) using gpt-4o-mini under a limited budget. The margins persist on harder suites (BigCodeBench-Instruct and Humanity's Last Exam) and transfer to multi-hop question answering (HotpotQA). Code available at `https://github.com/abcabdl/SMART`.

1 Introduction

LLM-powered agents, especially when organized into multi-agent systems (MAS), have shown strong potential for solving complex end-to-end tasks through collaboration and tool use [Hong et al., 2023, Liu et al., 2023, Xie et al., 2024]. As the emerging Agentic Web suggests, agent capabilities and tool ecosystems are expanding rapidly, pushing MAS into a regime with broader capabilities and increasingly heterogeneous candidate pools [Yang et al., 2025, Qin et al., 2024]. In this setting, the bottleneck is no longer only how to coordinate a fixed team [Kimi, 2026], but how to form the right team, assign the right responsibilities, and choose an appropriate interaction structure under strict efficiency constraints. This shift introduces two key challenges.

Challenge 1: Team formation in large, open agent pools. The first challenge is how to form a task-specific team from a large and heterogeneous candidate pool. Future systems may need to select from a large and open ecosystem of agents and tools, where many candidates overlap in function but differ in quality, specialization, latency, and cost [Yang et al., 2025]. ToolBench [Guo et al., 2024] already shows that effective tool use over tens of thousands of real-world APIs requires systematic retrieval and selection over large candidate sets.

✉Correspondence to: Weiwen Liu <wwliu@sjtu.edu.cn>.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Challenge 2: Online agent assignment under intermediate feedback. The second challenge is how to assign agents during execution when decisions must be made sequentially under partial feedback, including the selected agents and task-completion status. In this setting, search becomes a natural mechanism for online decision making. AFLOW [Zhang et al., 2025b], for example, uses MCTS to explore workflow spaces with execution feedback. However, existing search-based methods mainly optimize workflow structures or reasoning paths, rather than selecting among large candidate agent pools [Shang et al., 2025, Liu et al., 2023], and methods such as LATS [Zhou et al., 2024] and AFLOW [Zhang et al., 2025b] still rely on LLM-based generation or evaluation during search, which becomes costly as the candidate pool grows. It therefore remains unclear how to efficiently assign suitable agents to task-specific responsibilities from a massive candidate pool while balancing task utility, team complementarity, and inference cost.

To address these challenges, SMART (Scalable Multi-Agent Role-conditioned Teaming) formulates responsibility-to-agent assignment as a sequential and combinatorial decision problem. First, a schema-constrained topology planner determines a task-specific collaboration topology and candidate role set. Second, after constructing a diverse and relevant candidate set, an LLM-free MCTS fills each role online via simulated rollouts, capturing team-level compatibility without incurring LLM-based evaluation during search. Here "LLM-free" refers to this search stage only; LLMs still perform topology planning and agent execution. Third, we construct a large-scale library for controlled team formation over heterogeneous pools. Together, this design adapts team structure and agent assignment across tasks while improving both performance and efficiency.

Our contributions can be summarized as follows:

- **Dynamic & Lightweight Orchestration:** We propose a three-stage framework that decouples macro-architecture planning from micro-role filling, and introduce an LLM-free online MCTS [Kocsis and Szepesvári, 2006] for efficient role-to-agent assignment based on rollout feedback.
- **Large-Scale Ecosystem:** We build a routing-oriented agent-tool ecosystem comprising 300 task-specialized agents to study multi-agent composition and routing in open agent pools.
- **Empirical Effectiveness:** We show that the framework consistently outperforms strong baselines across reasoning and coding benchmarks, with margins that persist on harder suites and on multi-hop QA, and that the gains are stable across repeated runs. It consistently improves the cost-performance trade-off over compared baselines, and uses a step reward that correlates strongly with task success.

2 Related Work

Fixed-Topology Multi-Agent Systems. Early approaches to reasoning and multi-agent systems often rely on fixed collaboration patterns or predefined prompting strategies. Foundational methods such as Chain-of-Thought (CoT) [Wei et al., 2022] and Self-Consistency (SC) [Wang et al., 2023] improve single-agent reasoning, while methods like LLM-Debate [Du et al., 2024] use fixed ensembling and peer evaluation. Similarly, multi-agent frameworks such as CAMEL [Li et al., 2023], MetaGPT [Hong et al., 2023], and MacNet [Qian et al., 2025] organize collaboration through predefined roles or hard-coded communication structures. While these provide strong baselines, they are less suited to open and expanding ecosystems where team composition must be determined dynamically for each query.

Dynamic Orchestration and Role Planning. More recent work improves adaptivity by dynamically organizing agents, roles, or collaboration structures. AutoAgents [Chen et al., 2024] explores task-conditioned role generation, DyLAN [Liu et al., 2023] updates agent teams based on interaction history, AgentSquare [Shang et al., 2025] searches over modular MAS design spaces, and MasRouter [Yue et al., 2025] routes collaboration modes, roles, and models for query-dependent orchestration. These methods mainly focus on structural orchestration, whereas our work further addresses how to assign concrete agents to task-specific responsibilities from a massive candidate pool during execution.

Search-Based Optimization for Agentic Systems. Search has been widely used to optimize reasoning and workflow design in agentic systems. Language Agent Tree Search applies tree search

to reasoning and acting [Zhou et al., 2024], while methods such as GPTSwarm [Zhuge et al., 2024], ADAS [Hu et al., 2025], and AFLOW [Zhang et al., 2025b] optimize graph structures or reasoning paths through search or evolution. In contrast, we use MCTS not for workflow discovery, but for online responsibility-to-agent assignment under large candidate pools, with LLM-free rollouts for efficient search.

Large-Scale Tool and Agent Selection. As tool and agent libraries grow, retrieval and routing become increasingly important. Prior work ranges from large-scale tool retrieval over real-world APIs [Qin et al., 2024, Xie et al., 2024] to generalized routing across heterogeneous models and agents [Guo et al., 2025]. MAS-GPT [Ye et al., 2025] instead generates query-specific MAS implementations in a one-shot manner. Compared with these approaches, our method focuses on sequential team formation and responsibility-aware agent assignment guided by execution feedback, rather than one-shot selection of tools or system configurations.

3 Method

Figure 1 summarizes SMART as a closed-loop, three-stage orchestration framework. Given a user task, SMART first performs task-conditioned topology planning, which selects an appropriate collaboration structure and a compact set of functional roles. For each active role, SMART then constructs a role-conditioned candidate pool by retrieving relevant agents, re-ranking them for diversity, and applying a lightweight metadata-based unary score. Finally, an LLM-free MCTS module fills the current role by simulating downstream role-agent compatibility without invoking LLMs during search. After the selected agent executes, the system updates the workflow state and routes execution to the next role until the task is solved or the step budget is reached. This design decouples high-level collaboration planning from online role-to-agent assignment, preserving task-specific flexibility while keeping search efficient.

3.1 Notation and Problem Formulation

Let $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$ denote the agent pool, x the task description, and $\mathcal{R} = \{r_1, \ldots, r_M\}$ the role vocabulary including planner, builder, checker, refactor, and manager. At step t, the workflow executes role r_t and selects an agent a_t from a role-conditioned candidate set $\mathcal{C}^{(t)} \subseteq \mathcal{A}$. The runtime state is

$$s_t = (x,\ r_t,\ \mathcal{H}_t^a,\ \mathcal{H}_t^r,\ b_t,\ g_t), \tag{1}$$

where $\mathcal{H}_t^a = (a_0, \ldots, a_{t-1})$ and $\mathcal{H}_t^r = (r_0, \ldots, r_{t-1})$ record the agent and role histories. The artifact bag b_t stores task-family-specific fields such as draft code, normalized code, candidate answer, evidence points, supporting facts, verification, diagnostics, and final answer. The Boolean gap vector g_t is recomputed from b_t after each execution step.

3.2 Stage 1: Task-Conditioned Topology Planning

The first stage determines a task-specific macro-plan. Given task x and the predefined role vocabulary R, let π denote the topology planner, which queries a configured API meta-router under a JSON schema and returns a validated plan:

$$(\tau, R_{\text{cand}}, r_{\text{mgr}}, r_0, H_{\max}) := \pi(x),$$

where $\tau \in \{\texttt{single}, \texttt{centralized}, \texttt{decentralized}, \texttt{parallel}\}$ is the selected topology type, $R_{\text{cand}} \subseteq R$ is the activated role subset for task x, r_{mgr} is an optional manager role, r_0 is the entry role, and $H_{\max}$ is the step budget. The returned fields are schema-validated, and $H_{\max}$ is clamped to the runtime limit. The planner prompt and the role-conditioned execution prompts are given in Appendix C.

Compact role vocabulary. We use a small set of *functional* roles rather than many organization-specific titles. Recent multi-agent software systems already show that a few recurring functions are often sufficient: CODESIM uses three agents for planning, coding, and debugging, while RTADev uses five agents spanning requirements, architecture, planning, implementation, and testing [Islam et al., 2025, Liu et al., 2025]. SMART follows this abstraction to preserve the core stages of

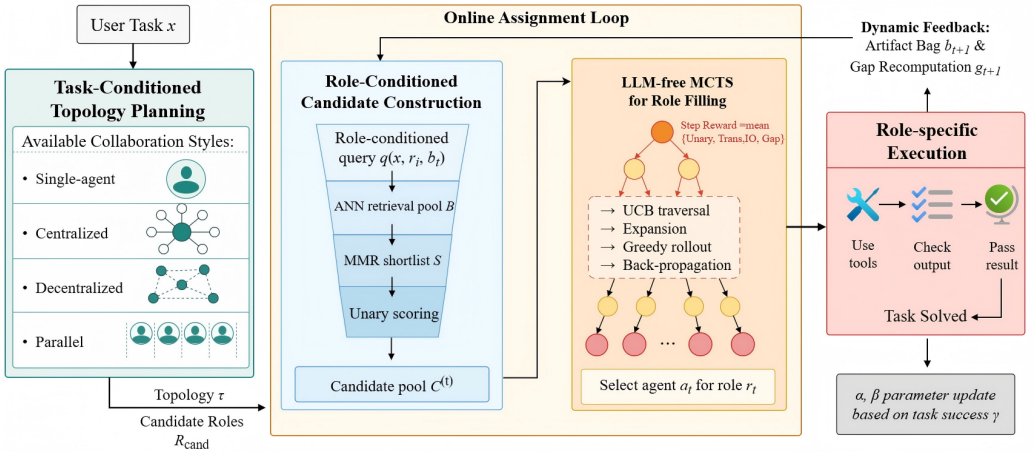


Figure 1: The overall dynamic orchestration framework of our proposed SMART

task completion while keeping role-conditioned retrieval and online assignment tractable in large heterogeneous agent pools.

3.3 Stage 2: Role-Conditioned Candidate Construction

For each active role r_t, we construct the candidate set $\mathcal{C}^{(t)}$ via a three-stage pipeline to obtain a compact set that is *relevant*, *diverse*, and *efficient* while remaining tractable as the agent pool size grows.

3.3.1 Role-conditioned Retrieval and MMR Re-ranking

We construct a role-conditioned query $\hat{q}_t := \hat{q}(x, r_t, b_t)$ from the task description, role descriptor, and recent artifact context. Approximate nearest-neighbor (ANN) retrieval [Arya et al., 1998] over normalized agent embeddings returns an initial pool B of size N using cosine similarity. Starting from $S_0 = \varnothing$, we then apply Maximal Marginal Relevance (MMR) [Carbonell and Goldstein, 1998] greedily for K rounds. At round j, we select

$$a_j^\star = \arg \max_{a \in B \setminus S_{j-1}} \left[\lambda \, s(\hat{q}_t, a) - (1-\lambda) \max_{b \in S_{j-1}} s(a, b) \right],$$

where the diversity penalty is defined as 0 when $S_{j-1} = \varnothing$, and we update $S_j = S_{j-1} \cup \{a_j^\star\}$. The final shortlist is $S = S_K$. Here $s(\cdot, \cdot)$ is cosine similarity and $\lambda \in [0, 1]$ controls the relevance-diversity trade-off. MMR runs in $O(NK)$, with a single interpretable knob λ, and is more efficient than DPP [Kulesza and Taskar, 2012], which requires $O(N^2 K)$ kernel decomposition per role.

3.3.2 Pre-MCTS Unary Scoring

From the MMR shortlist, SMART computes a metadata-only unary score for two purposes: it truncates the shortlist to the final candidate cap used by MCTS, and it serves as the unary-quality term in the step reward design.

$$\text{Unary}_t(a) = \text{norm}_S\Big(s(\hat{q}_t, a) + \tilde{m}(a, x, r_t) + \tilde{e}(a, x, r_t) - \tilde{c}(a) - \tilde{\ell}(a) \Big). \tag{2}$$

Here $s(\hat{q}_t, a)$ is the retrieval relevance term from the role-conditioned query, $\tilde{m}$ and $\tilde{e}$ are calibrated model-tier and function-band match bonuses, and $\tilde{c}$ and $\tilde{\ell}$ are calibrated cost and latency penalties. We keep the top K_f agents by unary score to form $C^{(t)}$, and reuse $\text{Unary}_t(a)$ as the unary-quality component in Eq. (6). The exact tier, band, penalty, and gap-cue rules are given in Appendix B.1.

Truncation before search is irreversible: an agent filtered out here cannot be recovered by MCTS. Two design choices limit this risk. The objective is the best completion of the *current* team rather than a single globally best agent, since $\hat{q}_t$ is conditioned on the live artifact context and MMR keeps complementary candidates from being crowded out by near-duplicates; and the step reward re-ranks the survivors by team-level signals rather than by retrieval score alone. Appendix B.1 gives details and evidence that the cap is not a binding bottleneck.

3.4 Stage 3: LLM-free MCTS for Online Role Filling

Given s_t and $\mathcal{C}^{(t)}$, MCTS selects a_t for role r_t. Search nodes store selected agent IDs, selected roles, current role, depth, accumulated reward, and the simulated gap vector. No LLM calls are made inside tree search or rollout.

3.4.1 Step Reward Design

To guide MCTS without LLM-based evaluation, we define a lightweight step reward with four components: unary quality, transition compatibility, I/O-chain compatibility, and artifact-gap reduction. Each component targets a distinct failure mode of online team formation: choosing the wrong specialist, stacking strong individuals into a weak team, broken handoffs between roles, and lack of progress on the task.

Unary quality (wrong specialist). Is the candidate suitable for this role in isolation? This is the classic person–role fit question. The unary component is the normalized pre-MCTS score $\mathrm{Unary}_t(a)$ from Eq. (2).

Transition compatibility (strong individuals, weak team). Does the candidate combine well with the agents already selected? Studies of human teams show that group performance is not explained by the average or maximum ability of its members alone [Woolley et al., 2010]; the transition term scores pairwise collaboration history so that search does not greedily stack individually strong but mutually ineffective agents. If previous agents exist, the transition component is the mean of Beta posterior means over previous-to-current agent pairs:

$$\mathrm{Trans}_t(a) = \frac{1}{t}\sum_{k<t} \frac{\alpha_{a_k \to a}}{\alpha_{a_k \to a} + \beta_{a_k \to a}}. \tag{3}$$

We average over all previous agents to encourage global team compatibility rather than only local handoff quality. Unobserved pairwise transitions are initialized with Beta(1,1). When persistent cross-instance updates are disabled, the transition term defaults to 0.5.

I/O-chain compatibility (broken handoffs). Can the candidate consume the predecessor's output and produce what the next role needs? This operationalizes the management of producer–consumer dependencies in coordination theory [Malone and Crowston, 1994]: a capable agent is still ineffective if it cannot consume the previous role's output. For adjacent agents, the raw I/O score is defined as the mean of three Jaccard overlaps over output formats, modalities, and tool tags.

$$\mathrm{IO}_t(a) = \frac{1}{3}(J(\mathcal{F}_{t-1}, \mathcal{F}_t) + J(\mathcal{M}_{t-1}, \mathcal{M}_t) + J(\mathcal{T}_{t-1}, \mathcal{T}_t)). \tag{4}$$

Here $\mathcal{F}$, $\mathcal{M}$, and $\mathcal{T}$ denote output-format, modality, and tool-tag metadata sets. If no previous agent exists, the I/O component uses the neutral value 0.5.

Artifact-gap reduction (no progress). Does the candidate close what is still missing from the task state rather than repeat completed work? This follows means–ends analysis [Newell and Simon, 1972]: an operation is chosen by the remaining difference between the current and goal states, where here the operation is an agent assignment and the difference is the unresolved gap vector. Let $\mathcal{J}_t = \{j : g_{t,j} = 1\}$ be the active unresolved gaps. This component scores whether candidate metadata and role cues can close those gaps:

$$\mathrm{Gap}_t(a, r_t) = \frac{1}{|\mathcal{J}_t|}\sum_{j \in \mathcal{J}_t} \Delta_j(a, r_t), \tag{5}$$

where $\Delta_j(a, r_t) \in [0, 1]$ is a deterministic capability match computed from role tags, tool tags, produced artifact fields, and task-family rules. During simulation, the predicted next unresolved gap is marked resolved when this score reaches the fixed threshold 0.55. Appendix B.1 lists the capability matching table.

Combined reward. We aggregate the active components by their mean:

$$R_t(a, r_t) = \frac{1}{|\Omega_t|}\sum_{z \in \Omega_t} z, \qquad \Omega_t \subseteq \{\mathrm{Unary}, \mathrm{Trans}, \mathrm{IO}, \mathrm{Gap}\}. \tag{6}$$

This default mean aggregation is referred to as `mean_active`. In the default runtime configuration, all four components are enabled whenever defined. Before the first handoff, missing transition or I/O terms fall back to the neutral value 0.5. The ablations in Table 4 confirm that each term matters (removing Gap costs -2.42/-3.52 Pass@1 on HumanEval/MBPP and removing IO costs -1.64/-1.47), and Appendix F.1 shows that the reward-side constants sit on a plateau rather than a knife-edge.

3.4.2 Tree Search Procedure

For each active role r_t, we run a lightweight MCTS over the role-conditioned candidate set $\mathcal{C}^{(t)}$ and score each candidate by its discounted proxy return Q_t. At each search node, traversal chooses the next child to visit by maximizing the upper confidence bound (UCB):

$$Q_t(a) \triangleq \mathbb{E}\left[\sum_{h=0}^{H-1} \gamma^h R_{t+h} \,\middle|\, a_t = a\right], \quad \mathrm{UCB}(c) = \frac{W(c)}{N(c)} + C_{\mathrm{ucb}}\sqrt{\frac{\ln N(\mathrm{p}(c))}{N(c)}}, \tag{7}$$

Here $W(c)$ and $N(c)$ are the accumulated return and visit count of node c, and $p(c)$ denotes its parent. Expansion appends one candidate and predicts the next role; rollout is greedy under Eq. (6) without LLM calls, and discounted returns are back-propagated along the visited path. After exhausting the search budget, we select the root child with the largest visit count.

$$a_t^* = \arg\max_{a \in \mathcal{C}^{(t)}} N\big(c_t(a)\big). \tag{8}$$

Detailed expansion, rollout, and back-propagation rules are provided in Appendix B.2.

3.5 Execution and Online Update

After selecting (r_t, a_t^*), the role is executed under a role-specific I/O contract. The output is normalized into b_{t+1}, the gap vector g_{t+1} is recomputed, and the next role is chosen based on planner handoff fields, agent-provided `next_role`, optional router output, and topology constraints. Execution stops when a completion check succeeds, no next role is available, or $H_{\max}$ is reached.

Persistent posterior updates are controlled by configuration. The main evaluation scripts disable cross-instance cache and cross-instance posterior updates by default. When enabled, the code updates role-level and pair-level Beta posteriors with the terminal binary signal $Y \in \{0, 1\}$:

$$\alpha_{a_k \to a_t} \leftarrow \alpha_{a_k \to a_t} + Y\rho, \qquad \beta_{a_k \to a_t} \leftarrow \beta_{a_k \to a_t} + (1 - Y)\rho. \tag{9}$$

The main executed sequence is updated with confidence $\rho = 1.0$, while shadow selections are updated with lower confidence when shadow feedback is recorded. Within an episode, selection caching can reuse role decisions keyed by topology, role, step, and task hash; it is separate from the persistent cross-instance bandit database.

4 Experiments

4.1 Experimental setup

Datasets and benchmarks. We conduct the main comparison on HumanEval [Chen et al., 2021], MBPP [Austin et al., 2021], GSM8K [Cobbe et al., 2021], and MATH [Hendrycks et al., 2021], covering different tasks of reasoning and problem-solving. For the MATH benchmark, we follow [Zhang et al., 2025a] in selecting a harder subset (level 5). To test harder and more open-ended settings, we additionally evaluate on HotpotQA in the distractor setting [Yang et al., 2018], BigCodeBench-Instruct [Zhuo et al., 2025], and the text-only math questions of Humanity's Last Exam (HLE) [Phan et al., 2025] (Section 4.3.2; protocol in Appendix D).

Baselines. We compare against two categories of baselines: (1) fixed or manually designed reasoning/workflow methods, including Chain-of-Thought (CoT) [Wei et al., 2022], ComplexCoT [Fu et al., 2023], Self-Consistency (SC) [Wang et al., 2023], LLM-Debate [Du et al., 2024], LLM-Blender [Jiang et al., 2023], DyLAN [Liu et al., 2023], MacNet [Qian et al., 2025]; (2) Autonomous workflows: AutoAgents [Chen et al., 2024], AgentSquare [Shang et al., 2025], GPTSwarm [Zhuge et al., 2024],

Table 1: **Performance Comparison of Agentic Workflow Methods.** The best results are highlighted in bold, and second-best results are underlined. LLMs used for each method are also listed for fair comparison. The LLM pool contains gpt-4o-mini, claude-3.5-haiku, gemini-1.5-flash, and llama-3.1-70b, and agents can be routed to different LLMs based on current needs.

Method	LLM	Math Reasoning		Code Generation		Average
		GSM8K	MATH	HumanEval	MBPP	
Vanilla	gpt-4o-mini	87.45	46.29	87.08	71.83	73.16
Top-1 Embedding	gpt-4o-mini	87.67↑0.22	47.32↑1.03	87.78↑0.70	72.73↑0.90	73.88↑0.72
CoT	gpt-4o-mini	87.10↓0.35	46.40↑0.11	88.13↑1.05	71.83↓0.00	73.37↑0.21
ComplexCoT	gpt-4o-mini	86.89↓0.56	46.53↑0.24	87.49↑0.41	72.36↑0.53	73.32↑0.16
SC(CoT×5)	gpt-4o-mini	87.57↑0.12	47.91↑1.62	88.60↑1.52	73.60↑1.77	74.42↑1.26
LLM-Debate	gpt-4o-mini	89.47↑2.02	48.54↑2.25	88.68↑1.60	70.29↓1.54	74.25↑1.09
LLM-Blender	gpt-4o-mini	88.35↑0.90	46.92↑0.63	88.80↑1.72	77.05↑5.22	75.28↑2.12
DyLAN	gpt-4o-mini	89.98↑2.53	48.63↑2.34	90.42↑3.34	77.30↑5.47	76.58↑3.42
MacNet	gpt-4o-mini	87.95↑0.50	45.18↓1.11	84.57↓2.51	65.28↓6.55	70.75↓2.41
AutoAgents	gpt-4o-mini	87.69↑0.24	45.32↓0.97	87.64↑0.56	71.95↑0.12	73.15↓0.01
AgentSquare	gpt-4o-mini	87.62↑0.17	48.51↑2.22	89.08↑2.00	78.46↑6.63	75.92↑2.76
GPTSwarm	gpt-4o-mini	89.14↑1.69	47.88↑1.59	89.32↑2.24	77.43↑5.60	75.94↑2.78
ADAS	gpt-4o-mini	86.12↓1.33	43.18↓3.11	84.19↓2.89	68.13↓3.70	70.41↓2.75
AFLOW	gpt-4o-mini	91.16↑3.71	51.28↑4.99	90.93↑3.85	81.67↑9.84	78.76↑5.60
MasRouter	LLM-Pool	<u>93.80</u>↑6.35	56.42↑10.13	90.62↑3.54	84.00↑12.17	81.21↑8.05
EvoFlow	gpt-4o-mini	92.90↑5.45	57.70↑11.41	92.85↑5.77	84.50↑12.67	81.99↑8.83
SMART	gpt-4o-mini	93.64↑6.19	<u>61.73</u>↑15.44	<u>93.89</u>↑6.81	<u>87.68</u>↑15.85	<u>84.24</u>↑11.08
SMART	LLM-Pool	**94.78**↑7.33	**62.34**↑16.05	**94.65**↑7.57	**88.86**↑17.03	**85.15**↑11.99

All rows are single runs under the same protocol. Five independent reruns of SMART (both settings) and MasRouter (Appendix F.2) give run-to-run standard deviations of 0.18–1.39 pp and means within 0.5 pp of the values shown.

ADAS [Hu et al., 2025], AFLOW [Zhang et al., 2025b], EvoFlow [Zhang et al., 2025a], MasRouter [Yue et al., 2025]. We also include a lightweight retrieval baseline Top-1 Embedding, which encodes the task text and role descriptor into a query embedding and selects the agent whose embedding has the highest cosine similarity to that query. All hyperparameters and how they were chosen are listed in Appendix F.

LLM Backbones. We conduct main experiments using gpt-4o-mini [OpenAI, 2024b] or an LLM pool containing gpt-4o-mini, claude-3.5-haiku [Anthropic, 2024], gemini-1.5-flash [Google, 2024], and llama-3.1-70b [Grattafiori et al., 2024]. We also use gpt-4o [OpenAI, 2024a] as router for some ablation experiments. Decoding temperatures are assigned per function family a priori (0.0 for topology planning; 0.1–0.6 for execution functions, from deterministic verification to exploratory generation) and were not tuned on test outcomes; the full table is given in Appendix C.3.

Fairness of comparison. All methods are matched on backbone, test split, tool-function classes, and candidate pool; the only asymmetry in training data favors the baselines (EvoFlow and MasRouter use benchmark instances, SMART does not). Appendix A.5 gives the details.

4.2 Agent Ecosystem

To study multi-agent composition at scale, we construct an ecosystem of 300 task-specialized agents across code generation and mathematical reasoning, each equipped with 5 LLM-generated executable tools. Each agent $a \in \mathcal{A}$ is represented by a structured metadata card that stores the signals used for candidate retrieval, interface compatibility scoring, and efficiency-aware selection. This metadata serves as the interface between the agent and our framework, supporting Stage 2 retrieval and Stage 3 LLM-free reward computation. The ecosystem is built through a programmatic synthesis pipeline; construction details are listed in Appendix A.

Table 2: Results on harder and open-ended suites. All methods share the same frozen 300-agent pool and the same backbone within each suite (gpt-4o-mini for HotpotQA and BigCodeBench-Instruct; gpt-5-mini with `reasoning_effort=medium` for HLE). "GPT-4o router" selects from the same pool with an LLM. Cost is the API cost per query.

Method	**HotpotQA (distractor, 500)**			**BigCodeBench-Instruct (200)**		**HLE Math (300)**	
	Ans. F1	Joint F1	Cost	Pass@1	Cost	Acc.	Cost
Single agent + tools	79.8	59.7	\$0.0016	51.0	\$0.0048	18.33	\$0.023
GPT-4o router	81.0	62.6	\$0.0044	54.5	\$0.0074	21.67	\$0.065
SMART (LLM-free MCTS)	**86.5**	**69.3**	\$0.0028	**62.5**	\$0.0058	**29.33**	\$0.041

4.3 Overall Effectiveness

4.3.1 Main Results

Table 1 shows that SMART consistently improves over same-backbone baselines on all four benchmarks, and the LLM-Pool variant further improves absolute performance. Relative to EvoFlow, SMART (gpt-4o-mini) improves GSM8K by +0.74, MATH by +4.03, HumanEval by +1.04, and MBPP by +3.18. Relative to Vanilla, the gains are +6.19, +15.44, +6.81, and +15.85, respectively. Notably, the lightweight retrieval baseline Top-1 Embedding remains far below SMART, indicating that retrieval alone is insufficient without execution-aware search. We also provide a detailed case study in Appendix E illustrating how SMART avoids capability-mismatched agent assignments and recovers the correct execution path on a representative HumanEval task, while Appendix A.4 shows that SMART remains effective under moderate metadata noise. The gains are stable across repeated runs: over five independent reruns, every SMART (LLM-Pool) run exceeds every MasRouter run on all four benchmarks (exact two-sided Mann–Whitney $p = 0.008$ per benchmark), and all five SMART (gpt-4o-mini) runs exceed EvoFlow's reported value on every benchmark (one-sided sign test $p = 0.031$); Appendix F.2 reports the full statistics.

4.3.2 Harder and Open-Ended Benchmarks

SMART's margins widen on harder suites and transfer to multi-hop QA. To test whether the gains in Table 1 depend on the difficulty or the structure of math and code tasks, we evaluate three additional suites (Table 2) under one shared protocol (Appendix D): pre-registered subsets, labels and hidden tests visible only to the offline evaluator, no external retrieval or repair loops, one pinned backbone per suite, and the router and SMART selecting from the identical frozen 300-agent pool. On HotpotQA (distractor), where evidence gathering and verification intertwine and no test-style completion signal exists, SMART improves over the GPT-4o router by $+5.5$ Answer F1 and $+6.7$ Joint F1 at 36% lower cost; the largest margin is on Joint F1, i.e., evidence completeness, exactly the state the gap vector monitors from the team's own outputs. On BigCodeBench-Instruct the single agent drops to 51.0 Pass@1, yet SMART leads the router by $+8.0$ at 22% lower cost, a wider margin than on HumanEval/MBPP. HLE lies beyond the discriminative band of gpt-4o-mini, so we upgrade every method to gpt-5-mini; SMART then leads the router by $+7.7$ accuracy at 37% lower cost, the largest relative gain of any suite. That the pattern survives a simultaneous change of task family and backbone indicates that the comparison measures relative orchestration ability rather than a property of one backbone or benchmark family.

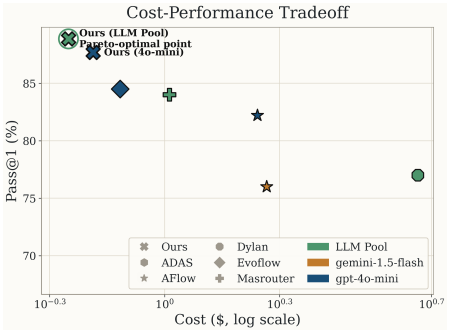
4.3.3 Performance under Heterogeneous Candidate Pools

Table 3: Performance on MBPP tasks under different heterogeneity settings.

Method	**Homogeneous**	**Skill Hetero.**	**Full Hetero.**
Top-1 Unary	79.18	81.72	82.11
Ours	80.94	87.68	88.86

To isolate the source of our method's gains, we compare MCTS with Top-1 Unary under **Homogeneous** agents (same model/capabilities), **Skill-Heterogeneous** agents (same model, varied capabilities), and **Full Heterogeneous** agents (varied specialization and backbone).

Table 3 shows modest gains in the homogeneous pool ($+1.76$), but larger gains under skill ($+5.96$) and full heterogeneity ($+6.75$), where specialization, backbone capability, and cost jointly affect routing.

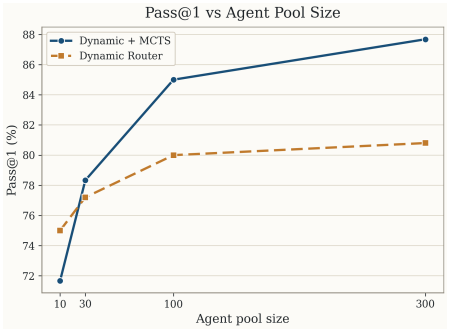


(a) Cost–performance trade-off on MBPP.

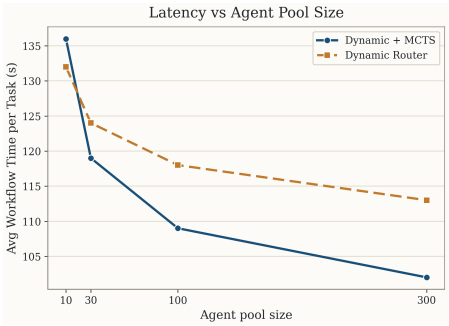


(b) Step reward vs. task success on MBPP.

Figure 2: **Cost-effectiveness and reward validity on MBPP.** Left shows our method achieves a stronger Pareto trade-off between cost and Pass@1. Right shows the step reward used in search is well aligned with actual task success.



(a) Pass@1 (%) vs. agent pool size.



(b) Avg. workflow time (s) vs. agent pool size.

Figure 3: Scalability on complex MBPP tasks (pool size 10–300). Dynamic-MCTS increasingly outperforms Dynamic-Router-GPT-4o at larger pool sizes while maintaining lower latency.

The three settings also form a coarse specialization-granularity axis: the finer the specialization in the pool, the larger the benefit of search-based assignment. Appendix F.1 reports sweeps over B, λ, C, K_f, and the reward-side constants, and Appendix A.6 varies tool coverage.

4.4 Efficiency and Scalability

4.4.1 Cost-Performance Tradeoff

We compute cost from input/output tokens for planner and executors; local MCTS rollouts and deterministic tools incur no API cost. Figure 2a records the overall cost ($) on MBPP and Pass@1 of different methods, showing that our method positions itself as the top performer on the cost-performance Pareto front, offering the highest Pass@1 while maintaining cost efficiency. The LLM-Pool variant boosts accuracy without increasing costs, demonstrating efficient model allocation via MCTS.

4.4.2 Scalability to Candidate Pool Size

We compare Dynamic-MCTS with Dynamic-Router-GPT-4o, which uses the same dynamic workflow but an online gpt-4o router, as the MBPP candidate pool scales from 10 to 300 agents. Here, we use MBPP tasks for which the planner determines that team collaboration is required. Figure 3 shows that the methods are comparable at small pool sizes, but MCTS pulls ahead from size 30 and keeps a growing gap at 300. It also has lower latency at pool sizes ≥ 30, consistent with better specialist matching and fewer retries. Appendix F.3 decomposes the wall-clock time along this sweep: total selector overhead grows only from 66 to 138 ms per step for a $30\times$ larger pool, while end-to-end time per task falls from 136 to 102 seconds because richer pools reduce LLM retries.

4.5 Search Signal and Component Analysis

4.5.1 MCTS Proxy Reward Predicts Task Success

A core assumption of our LLM-free MCTS is that the lightweight step reward R_t (Eq. 6) predicts true execution success. Figure 2b shows a positive correlation with binned success rate (r=0.814): low-reward bins ($\approx$0.3–0.5) reach only 0.3–0.4 pass rate, while the highest-reward bin (mean $\approx$0.9) reaches 1.0.

4.5.2 Ablation Studies

We ablate both stages. For topology planning, replacing the dynamic planner with a fixed Builder $\rightarrow$ Checker $\rightarrow$ Builder pipeline lowers HumanEval Pass@1 from 93.89 to 91.47 under MCTS selection and from 89.92 to 88.37 under the GPT-4o router, so both stages contribute independently. For agent selection, Table 4 shows that random or unary-only selection is weaker than MCTS, that the gap-aware and I/O-aware reward components contribute meaningfully, and that the LLM-free selector is over an order of magnitude faster per step than the GPT-4o router.

Table 4: Ablation on MCTS selection and reward design (latency: ms per selector step).

Selection Strategy	HumanEval		MBPP	
	Pass@1	Latency	Pass@1	Latency
Random-from-Retrieved	85.49	11	70.38	14
Top-1 Embedding	87.78	15	72.73	22
Top-1 Unary	89.14	28	81.72	35
MCTS w/o Gap	91.47	92	84.16	118
MCTS w/o I/O Chain	92.25	97	86.21	126
GPT-4o Router	89.92	1850	82.11	2140
MCTS (ours)	**93.89**	105	**87.68**	138

5 Limitations

Our work has four main limitations. First, the 300-agent ecosystem may still be small compared to future agent pools. Second, the ecosystem is synthetic: it uses LLM-generated executable tools, not live third-party services from an open Agentic Web. This controlled setup isolates routing behavior, but omits live API heterogeneity, service drift, authentication constraints, and tool failures; long-horizon web tasks, multimodal tasks, and live tool-use settings remain future work. Third, the step reward requires that execution exposes some checkable notion of remaining work. Math, code, and multi-hop QA do, but tasks with no verifiable intermediate state (e.g., free-form creative writing) deactivate the gap term, and since success itself cannot then be verified, evaluation is ill-defined for any method; truly subjective tasks are therefore out of scope. Fourth, retrieval truncation is irreversible, although the K_f sweep in Appendix F.1 suggests it is not a binding bottleneck.

6 Conclusion

We introduced SMART, a dynamic multi-agent orchestration framework for large-scale agent ecosystems. SMART decouples task-conditioned topology planning, role-conditioned candidate construction, and LLM-free MCTS-based role-to-agent assignment, enabling flexible and efficient team formation as agent libraries grow. Across reasoning, code-generation, and multi-hop QA benchmarks, SMART improves performance and cost-efficiency, with margins that widen on harder suites and are stable across repeated runs; ablations and scaling results further validate the contributions of dynamic planning and MCTS-based selection.

Acknowledgments and Disclosure of Funding

The work is supported by National Natural Science Foundation of China 62502310.

References

Anthropic. Introducing computer use, a new claude 3.5 sonnet, and claude 3.5 haiku. Anthropic News, October 2024. URL https://www.anthropic.com/news/3-5-models-and-computer-use.

Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 335–336, 1998.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje Karlsson, Jie Fu, and Yemin Shi. Autoagents: a framework for automatic agent generation. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pages 22–30, 2024.

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.

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. In *Forty-first international conference on machine learning*, 2024.

Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. Complexity-based prompting for multi-step reasoning. In *11th International Conference on Learning Representations, ICLR 2023*, 2023.

Google. Gemini breaks new ground with a faster model, longer context, ai agents and more. Google Blog, May 2024. URL https://blog.google/innovation-and-ai/products/google-gemini-update-flash-ai-assistant-io-2024/.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Xiyu Guo, Shan Wang, Chunfang Ji, Xuefeng Zhao, Wenhao Xi, Yaoyao Liu, Qinglan Li, Chao Deng, and Junlan Feng. Towards generalized routing: Model and agent orchestration for adaptive and efficient inference. *arXiv preprint arXiv:2509.07571*, 2025.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 11143–11156, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5113–5139, 2025.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14165–14178, 2023.

Kimi. Kimi k2.6: Advancing open-source coding. `https://www.kimi.com/blog/kimi-k2-6`, 2026.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

Alex Kulesza and Ben Taskar. Determinantal point processes for machine learning. *Foundations and Trends® in Machine Learning*, 5(2-3):123–286, 2012.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.

Jie Liu, Guohua Wang, Ronghui Yang, Jiajie Zeng, Mengchen Zhao, and Yi Cai. Rtadev: Intention aligned multi-agent framework for software development. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 1548–1581, 2025.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170*, 2023.

Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

Allen Newell and Herbert A. Simon. *Human Problem Solving*. Prentice-Hall, 1972.

OpenAI. Hello gpt-4o. OpenAI Blog, May 2024a. URL `https://openai.com/index/hello-gpt-4o/`.

OpenAI. Gpt-4o mini: advancing cost-efficient intelligence. OpenAI Blog, July 2024b. URL `https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/`.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Sean Shi, Michael Choi, Anish Agrawal, Arnav Chopra, et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. In *The Thirteenth International Conference on Learning Representations*, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations*, 2024.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. In *The Thirteenth International Conference on Learning Representations*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Anita Williams Woolley, Christopher F. Chabris, Alex Pentland, Nada Hashmi, and Thomas W. Malone. Evidence for a collective intelligence factor in the performance of human groups. *Science*, 330(6004):686–688, 2010.

Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, et al. Openagents: An open platform for language agents in the wild. In *First Conference on Language Modeling*, 2024.

Yingxuan Yang, Mulei Ma, Yuxuan Huang, Huacan Chai, Chenyu Gong, Haoran Geng, Yuanjian Zhou, Ying Wen, Meng Fang, Muhao Chen, et al. Agentic web: Weaving the next web with ai agents. *arXiv preprint arXiv:2507.21206*, 2025.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2369–2380, 2018.

Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. Mas-gpt: Training llms to build llm-based multi-agent systems. In *Forty-second International Conference on Machine Learning*, 2025.

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)*, pages 15549–15572, 2025.

Guibin Zhang, Kaijie Chen, Guancheng Wan, Heng Chang, Hong Cheng, Kun Wang, Shuyue Hu, and Lei Bai. Evoflow: Evolving diverse agentic workflows on the fly. *arXiv e-prints*, pages arXiv–2502, 2025a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, pages 62138–62160, 2024.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2406.15877.

A Large-Scale Ecosystem Construction

A.1 Agent Pool Generation

To rigorously evaluate multi-agent routing at scale, we construct a large ecosystem of specialized agents through a programmatic synthesis pipeline rather than manual authoring. To keep the benchmark self-contained and reproducible, we synthesize tool implementations offline with gpt-4o-mini and freeze the resulting registry before evaluation. Tool generation never has access to evaluation instances or external web content. The pipeline first plans benchmark-conditioned themes and role-aware profiles via prompt-based generation, and then instantiates each profile as an AGENTCARD with explicit capability metadata, including role tags, tool bindings, and cost/latency priors. In parallel, we incorporate a deterministic generator to ensure controlled scaling and strict reproducibility.

The comprehensive 300-agent setting is achieved through configuration-level scaling rather than architectural modifications, formalized as

$$N_{\text{agents}} = |\mathcal{B}| \times A_{\text{per-benchmark}}$$

By setting the number of benchmark families to $|\mathcal{B}| = 4$, the number of agents per benchmark to $A_{\text{per-benchmark}} = 75$, and the number of tools per agent to $T_{\text{per-agent}} = 5$, we obtain an ecosystem comprising 300 specialized agents. The four benchmark families are HUMANEVAL, MBPP, GSM8K, and MATH, which jointly cover two broader task regimes: code generation and mathematical reasoning. Importantly, benchmark conditioning is used only at construction time to synthesize diverse capability profiles rather than to handcraft agents for individual instances. At evaluation time, all synthesized agents are registered in a single shared pool, and the router retrieves and selects collaborators from this global pool for each query. Thus, the benchmark labels serve as capability-level metadata for large-scale pool construction, not as per-instance manual assignment rules.

Family-level alignment. Instance-level leakage is excluded by construction: tools are synthesized without access to evaluation instances, answers, execution feedback, or web content; the registry is frozen before evaluation; and at test time all agents sit in one shared pool with no per-instance assignment. What remains is family-level conditioning, which provides, per task domain, the capability categories a domain plausibly needs (solving, verification, normalization, formatting), i.e., the same classes of executable tools the baselines use, partitioned finely across 300 agents to instantiate the large-pool setting. Three observations put this alignment in context. (i) It is weaker than the alignment of the strongest baselines: EvoFlow evolves its workflows on benchmark training/validation instances with execution feedback and MasRouter trains its router on benchmark data, whereas SMART's ecosystem is built from the task family alone. (ii) Varying only the selection strategy over the same pool spans 85.49 to 93.89 Pass@1 on HumanEval and 70.38 to 87.68 on MBPP (Table 4); pool–test alignment alone cannot produce this gap. (iii) Performance does not hinge on finely tuned metadata: corrupting the metadata of 10% of agents costs 2.35–3.23 pp (Appendix A.4). The same frozen pool is also used unchanged on HotpotQA, BigCodeBench-Instruct, and HLE (Section 4.3.2), none of which conditioned its construction.

This ecosystem is intentionally constructed rather than web-collected. The registered tools are executable artifacts generated by the construction pipeline and exposed through stable interfaces; they should not be interpreted as live third-party services from an open Agentic Web. The benefit of this design is controlled scale: agent metadata, tool coverage, costs, latencies, and task-family capabilities can be varied without confounding changes from external services. The cost is realism: the benchmark does not model live API authentication, rate limits, endpoint drift, or provider outages.

A.2 Tool Ecosystem

Our tool ecosystem is assembled through an LLM-based generation pipeline that produces executable tools with stable interfaces and structured metadata. These LLM-generated tools include explicit input/output signatures, interface requirements, and executable implementations. Task-specific generators enforce non-overlapping capability assignments and automatically produce the corresponding tool manifests and registration entries.

The tool taxonomy is strictly metadata-driven:

- **Domain Tags:** Indicate the task domain (e.g., `code-generation`).
- **Role Tags:** Define the functional responsibility (e.g., `planner`, `checker`, `builder`, `refactor`).
- **Tool Tags:** Provide fine-grained capability and stage markers (e.g., `analyze`, `model`, `solve`, `verify`).

Invocation is standardized as `run(inputs: dict) -> dict`, and each tool is executed from a registry-stored codebase via a sandboxed executor with strict timeout and module controls. This provides a uniform executable interface for retrieval-time composition and runtime substitution.

A.3 Agent Metadata Schema

An abbreviated, benchmark-neutral `AgentCard` instance is presented below. The example avoids dataset- or task-id-specific names; in the released registry, the same fields are filled by the synthesis pipeline and linked to executable tool entries.

Listing 1: Example Agent Card Schema (JSON)

```
{
  "id": "agent-evidence-checker-017",
  "name": "EvidenceVerifier-Precise-Std",
  "kind": "agent",
  "version": "2.0",
  "updated_at": "2026-04-06T10:51:33Z",
  "domain_tags": ["evidence-grounding", "general-reasoning"],
  "role_tags": ["checker", "refactor"],
  "modalities": ["text"],
  "output_formats": ["json"],
  "permissions": ["read"],
  "cost_tier": "low",
  "latency_tier": "fast",
  "reliability_prior": 0.84,
  "description": "Verifier for claim checking, evidence matching, and
     answer formatting.",
  "tools": [
    {
      "id": "evidence-claim-extract",
      "stage": "analyze",
      "capability_kind": "claim-extractor",
      "consumes": ["task", "context"],
      "produces": ["claims", "diagnostics"],
      "tool_tags": ["evidence", "claim-extraction", "analyze"]
    },
    {
      "id": "evidence-support-check",
      "stage": "verify",
      "capability_kind": "support-verifier",
      "consumes": ["claims", "context"],
      "produces": ["verification", "supporting_spans"],
      "tool_tags": ["evidence", "verification", "support-check"]
    },
    {
      "id": "evidence-contradiction-scan",
      "stage": "verify",
      "capability_kind": "consistency-checker",
      "consumes": ["claims", "context", "verification"],
      "produces": ["diagnostics", "conflict_flags"],
      "tool_tags": ["evidence", "consistency", "verify"]
    },
    {
      "id": "evidence-answer-format",
      "stage": "format",
      "capability_kind": "answer-formatter",
```

```
      "consumes": ["candidate_answer", "verification"],
      "produces": ["final_answer"],
      "tool_tags": ["format", "answer"]
    },
    {
      "id": "evidence-output-audit",
      "stage": "audit",
      "capability_kind": "output-auditor",
      "consumes": ["final_answer", "supporting_spans", "diagnostics"],
      "produces": ["audit_report"],
      "tool_tags": ["audit", "evidence", "final-check"]
    }
  ],
  "embedding_text": "evidence verifier checker support grounding
     consistency answer audit"
}
```


A.4 Robustness to Noisy Agent Metadata

Experimental setup. We further evaluate the robustness of SMART under corrupted agent metadata on MBPP. We keep the evaluation pipeline unchanged and perturb only the structured metadata stored in the agent registry. Specifically, **Clean** denotes the original metadata setting without perturbation; **TagFlip-10%** randomly corrupts the `domain_tags` or `role_tags` of 10% of agents; **FakeIO-10%** injects mismatched `modalities` or `output_formats` into 10% of agents; and **FullMeta-10%** applies broader corruption over semantic and I/O-related metadata fields. We report Pass@1 on MBPP under each setting.

Table 5: Pass@1 (%) on MBPP under metadata corruption.

Method	Clean	TagFlip-10%	FakeIO-10%	FullMeta-10%
SMART	**87.68**	**85.04**	**85.33**	**84.45**

Results. Table 5 shows that SMART remains stable under moderate metadata noise. Starting from 87.68 Pass@1 in the clean setting, performance decreases to 85.04 under **TagFlip-10%**, 85.33 under **FakeIO-10%**, and 84.45 under **FullMeta-10%**. The absolute performance drop is limited to 2.64–3.23 % across all perturbation settings, indicating that the routing framework does not rely on perfectly clean metadata to function effectively.

Analysis. Two findings are worth noting. First, the overall degradation is modest, which suggests that SMART is reasonably robust to partial corruption in the agent registry. Even when 10% of metadata entries are perturbed, the system retains most of its original performance, implying that routing decisions are not determined solely by any single metadata field. Second, **FullMeta-10%** produces the largest drop, which is expected because it jointly affects both semantic tags and I/O-related metadata, thereby perturbing both candidate selection and inter-agent compatibility estimation. Overall, these results indicate that SMART is resilient to moderate metadata noise, while broader corruption across multiple metadata fields remains the most challenging case.

A.5 Comparison Protocol and Fairness

All methods are matched on the dimensions that determine task difficulty. *Model and data*: SMART (gpt-4o-mini) is compared with the gpt-4o-mini baselines, and SMART (LLM-Pool) with MasRouter on the identical four-model pool, all on the same public test splits. *Tools*: SMART uses the same broad classes of executable tool functions as the baselines, partitioned more finely across agent profiles; an agent is a role-conditioned prompt plus an allocation of these tools and a metadata card, and no external APIs or new tool classes are introduced. *Candidate pool*: the large pool is the research object; it adds no privileged data, external APIs, or per-query execution budget, and every pool-based selection baseline in Table 4 chooses from the identical pool. *Training data*: EvoFlow optimizes its workflows on benchmark training/validation instances with execution feedback and MasRouter

trains its router on benchmark data, whereas SMART's ecosystem, hyperparameter defaults, and temperature policy are fixed without access to any benchmark instance (Appendix A).

A.6 Robustness to Tool Coverage

Holding MBPP, the 300-agent pool, and all other settings fixed, we randomly remove tools from every agent (Table 6). Degradation is graceful: removing 20%/40% of all tools costs 1.50/3.92 pp. Notably, removing one of five tools from every agent (20% of all tools) hurts less than corrupting the metadata of only 10% of agents (Table 5): when a capability is absent, the selector can route around it through other agents in the pool, whereas corrupted metadata actively misleads selection. Wrong information is worse than missing capability.

Table 6: Pass@1 (%) on MBPP under reduced tool coverage (mean $\pm$ SD over random removals).

Tool coverage	Tools per agent	SMART Pass@1 (%)
100%	5/5	87.68
80%	4/5	86.18 ± 0.47
60%	3/5	83.76 ± 0.81

B Algorithmic Details

B.1 Reward Definition and Routing Rules

This appendix states the fixed rules behind Eq. (2) and Eq. (5). We keep the notation short and omit optional arguments when the active task and role are clear. These rules are fixed during evaluation and are not tuned per instance.

Model-tier match $m(a)$. Each active role is assigned a coarse difficulty bucket: LOW, MID, or HIGH. Builder, checker roles become HIGH when the query contains hard cues such as `multi-step`, `proof`, `optimize`, `edge case`, `security`, or when the current artifact has failed verification; manager, planner, and refactor default to LOW unless such cues are present; all remaining cases are MID. Agent tiers are read from the name suffix and mapped as `std`→LOW, `adv`→MID, and `pro`→HIGH. The match bonus is 1 for the same bucket, 0.5 for an adjacent bucket, and 0 otherwise. Unknown tiers receive 0.5.

Function-band match $e(a)$. Agent names also encode a function band, grouped into PRECISE, BALANCED, or CREATIVE. Builder, checker, and refactor target PRECISE; planner and manager target BALANCED; researcher targets CREATIVE. Precision cues such as `exact`, `strict`, `assert`, `verify`, `unit test`, or `bugfix` move the target one step toward PRECISE; exploration cues such as `research`, `explore`, `brainstorm`, `alternatives`, or `creative` move it one step toward CREATIVE. The band bonus is again 1 for a match, 0.5 for an adjacent band, and 0 otherwise.

Unary candidate score. For each role, the pre-normalized unary score combines the retrieval role score $q(a, r)$, the fixed bonuses $m(a)$ and $e(a)$, and small cost/latency penalties. Low or fast tiers receive no penalty, medium tiers receive a half penalty, high or slow tiers receive a full penalty, and missing values receive the midpoint. The resulting scalar is min–max normalized over the shortlisted candidates before entering Eq. (2).

Artifact-gap rule $\Delta_j(a)$. For every unresolved artifact gap j, the implementation stores a fixed cue set: preferred agent roles, tool stages, capability kinds, and produced artifact fields. Let $\mathrm{cov}_a(U)$ be the best coverage of any cue in set U among the tools owned by agent a, with empty cue sets scored as 0. The gap-reduction score is

$$\Delta_j(a) = \max\{\mathrm{role}(a, j), \mathrm{cov}_a(\mathcal{S}_j), \mathrm{cov}_a(\mathcal{C}_j), \mathrm{cov}_a(\mathcal{P}_j)\},$$

where $\mathrm{role}(a, j) = 1$ when any role tag of a is preferred for gap j, and 0 otherwise; the cue sets are listed in Table 7. The active gap score is the uniform mean over unresolved gaps. In simulation, only the primary next gap is closed, and only when this score reaches 0.55.

Table 7: Fixed artifact-gap cue table used by $\Delta_j(a, r)$. Dashes denote empty cue sets.

Gap	Preferred roles	Stage cues	Capability cues	Produced fields
missing entry point	planner, checker	analyze	signature extractor	entry point
missing draft code	builder, refactor	–	–	–
missing normalized code	builder, refactor	prepare	code normalizer	normalized code
missing verification	checker	verify, execute	public test runner, structure checker, answer verifier, support verifier	verification
has failure trace	checker, refactor	diagnose	traceback localizer	diagnostics
missing quantities	planner, checker	analyze	quantity extractor, signal profiler	–
missing relations	planner, checker	solve, support	relation calculator	–
missing candidate answer	builder, planner, refactor	–	–	–
missing format	checker, planner	format	answer formatter	formatted answer
missing structure parse	planner, checker	analyze	signal profiler	–
missing exact relation	planner, checker	solve, support	relation calculator	–
missing question decomposition	planner, checker	analyze	question decomposer	reasoning trace
missing evidence	planner, checker	retrieve	context indexer	evidence points
missing final answer	builder, checker	answer, verify	answer extractor, support verifier	final answer

On retrieval truncation. An agent filtered out before MCTS cannot be recovered later. Two design choices limit this risk. First, the objective is not a single globally best agent but the best completion of the *current* team: the query $\hat{q}_t$ is conditioned on the live artifact context, and MMR preserves diversity so that complementary candidates are not crowded out by near-duplicates of the top match. Second, the step reward re-ranks the survivors by team-level signals (Trans, IO, Gap) rather than by retrieval score alone. Empirically, the cap is not a binding bottleneck: enlarging K_f from 5 to 10 does not improve MBPP Pass@1 (Appendix F.1), which would be expected if highly suitable agents were being discarded.

B.2 Detailed Tree Search and Execution Procedure

Search state. At role step t, a search node stores the selected agent IDs, selected roles, current role, depth, accumulated discounted reward, and the simulated gap vector. Expanding a node appends one candidate from the current role-conditioned candidate cap and predicts the next role from topology edges, routing tables, gap-role preferences, and deterministic role-priority fallbacks.

Simulation and rollout. To keep MCTS rollouts lightweight and reproducible, SMART does not execute agents or tools inside simulation. Instead, it maintains a simulated gap state, a heuristic proxy for which task-family artifacts remain unresolved after each hypothetical role assignment. After simulating the current role, SMART updates this proxy gap state with a deterministic gap-transition rule and selects the next role by the following precedence: (1) stop if the rollout depth limit is reached or no unresolved simulated gap remains; (2) follow an explicit successor role if the topology or routing table specifies one; (3) otherwise map the primary unresolved simulated gap to its preferred role; and (4) if no such role is valid, use a fixed role-priority fallback among the remaining planner-allowed roles. Rollouts then greedily select the candidate with the largest current reward in Eq. (6), simulate artifact-gap closure using the same gap-transition heuristic, and accumulate

$$G_t = \sum_{h=0}^{H-1} \gamma^h R_{t+h}.$$

By default, the simulation closes the predicted next unresolved gap when the candidate's equal-weight gap-reduction score reaches 0.55, while all other transitions remain deterministic under the same task-family rules used during execution.

Tree policy. During traversal, MCTS selects the child with the largest Upper Confidence Bound:

$$c^* = \arg \max_{c \in \mathrm{Children}(v)} \mathrm{UCB}(c), \qquad \mathrm{UCB}(c) = \frac{W(c)}{N(c)} + C_{\mathrm{ucb}} \sqrt{\frac{\ln N(\mathrm{p}(c))}{N(c)}}.$$

Here $W(c)$ is the accumulated discounted return, $N(c)$ is the visit count, and C_{ucb} controls exploration.

Back-propagation and final decision. After each rollout, the return is back-propagated along the visited path:

$$W(c) \leftarrow W(c) + G_t, \qquad N(c) \leftarrow N(c) + 1.$$

After the search budget B is exhausted, the final agent is selected by visit count:

$$a_t^* = \arg\max_{a \in \mathcal{C}^{(t)}} N\big(c_t(a)\big).$$

Execution and early termination. Once (r_t, a_t^*) is selected, the role is executed under a role-specific input/output contract. Tool outputs are merged into the artifact bag, and task-family completion checks read normalized fields such as generated code, candidate answers, evidence points, verification results, and final answers. After each execution, SMART recomputes the gap vector from the artifact bag and checks whether the task is solved. If the current artifacts are sufficient, the episode terminates early; otherwise, SMART selects the next role and continues execution.

Posterior update and caching. Finally, SMART maintains lightweight in-episode selection caches and, when enabled, updates persistent posteriors after the episode ends. The workflow has an in-episode selection cache keyed by topology, role, step, and task hash. Persistent role and pair posteriors are loaded and updated only when cross-instance cache and posterior-update flags are enabled. When enabled, realized pairwise transition posteriors are updated using the binary terminal signal $Y \in \{0, 1\}$:

$$\alpha_{a_k \to a_t} \leftarrow \alpha_{a_k \to a_t} + Y\rho, \qquad \beta_{a_k \to a_t} \leftarrow \beta_{a_k \to a_t} + (1 - Y)\rho,$$

where $\rho = 1.0$ for the main executed sequence and a lower confidence is used for shadow feedback. The main benchmark evaluation entry points disable persistent cross-instance updates by default.

B.3 Overall algorithm

Algorithm 1 summarizes the SMART routing loop. The online phase alternates between candidate construction, MCTS-based role selection, execution, artifact-gap recomputation, and optional posterior updates.

Algorithm 1 SMART Routing Loop with Artifact-Gap-Aware MCTS

Require: Query x, agent pool $\mathcal{A}$, role vocabulary $\mathcal{R}$, budgets (N, K, K_f, B, H), MMR trade-off λ, penalties (μ, ν), discount γ, UCB constant C_{ucb}, optional cross-instance bandit database
Ensure: Executed team trace, final artifact bag, optional updated posteriors

$\pi(x) \leftarrow \textsc{PlanTopology}(x, \mathcal{R})$
Initialize histories $\hat{\mathbf{a}}, \hat{\mathbf{r}} \leftarrow \varnothing$ and artifact bag b_0
$g_0 \leftarrow \textsc{ComputeGap}(b_0), \quad r \leftarrow r_0, \quad t \leftarrow 0$
while $r \neq \varnothing$ **and** $t < \pi(x).H_{\max}$ **do**
  $q_r \leftarrow \textsc{BuildRoleQuery}(x, r, b_t)$
  $\mathcal{B}_r \leftarrow \text{ANN-}\textsc{TopN}(\mathcal{A}, q_r, N)$
  $S_r \leftarrow \text{MMR-}\textsc{TopK}(\mathcal{B}_r, K, \lambda)$
  for each $a \in S_r$ **do**
    $U_{\mathrm{raw}}(a) \leftarrow q(a, r) + 0.08m(a, x, r) + 0.06e(a, x, r) - \mu c(a) - \nu \ell(a)$
  end for
  $\mathcal{C}^{(t)} \leftarrow \textsc{TopKf}(S_r, U_{\mathrm{raw}}, K_f)$
  Initialize MCTS root with $(\hat{\mathbf{a}}, \hat{\mathbf{r}}, r, g_t)$
  for $b = 1$ to B **do**
    Traverse by UCB until an expandable node is reached
    Expand one untried candidate $a \in \mathcal{C}^{(t)}$
    $\Omega \leftarrow \{\mathrm{Unary}, \mathrm{Trans}, \mathrm{IO}, \mathrm{Gap}\}$ *(I/O is neutral before the first handoff)*
    $R \leftarrow |\Omega|^{-1} \sum_{z \in \Omega} z$ *(`mean_active` reward)*
    Simulate next gap vector and next role; greedily roll out to depth H
    Back-propagate discounted return to all visited nodes
  end for
  $a_t^\star \leftarrow \arg\max_{v \in \textsc{Children(root)}} N(v)$
  $o_t \leftarrow \textsc{ExecuteRole}(r, a_t^\star, b_t)$
  $b_{t+1} \leftarrow \textsc{MergeArtifact}(b_t, o_t), \quad g_{t+1} \leftarrow \textsc{ComputeGap}(b_{t+1})$
  Append $a_t^\star$ and r to histories
  if $\textsc{Complete}(b_{t+1}, g_{t+1})$ **then**
    $r \leftarrow \varnothing$
  else
    $r \leftarrow \textsc{SelectNextRole}(x, b_{t+1}, g_{t+1}, \pi(x))$
  end if
  $t \leftarrow t + 1$
end while
if cross-instance cache and posterior updates are enabled **then**
  Update role and pair Beta posteriors with terminal label Y
end if
return histories, final artifact bag, optional updated posteriors

C Prompts and implementation details

C.1 Topology planner and API interface

To ensure strict controllability of macro-architecture planning, the topology planner queries an API-based meta-router under a constrained JSON schema.

C.1-A Planner API Prompt

```
[System]
You are a topology planner. Decide the best agent topology for the task.
Return ONLY JSON with keys: topology, roles, manager_role, entry_role, max_steps.
If topology is ‘parallel‘, also include optional keys: parallel_width,
    parallel_roles,
aggregation_role, consensus_policy, branch_timeout, fail_fast.

[User]
Task: {task_text}
Available roles: {roles_json}
Topology options: single, centralized, decentralized, parallel.
IMPORTANT: Choose the topology based on coordination complexity.
Use ‘single‘ when one role can complete the task independently with minimal
    decomposition.
Use ‘centralized‘ when the task requires a manager to coordinate multiple
    specialized roles.
Use ‘decentralized‘ when coordination must be distributed across peers for better
     collaboration, rather than controlled by a single manager.
Use ‘parallel‘ when independent branches can run concurrently and then be merged
    by an aggregation role.

Field rules:
‘roles‘ must be an ordered non-empty subset of Available roles.
Use the minimal role set needed for the task.
‘manager_role‘ is null except for ‘centralized‘, where it should be the
    coordinating role, usually ‘planner‘ if available.
‘entry_role‘ is the first role to execute; for ‘centralized‘, use the manager
    role.
‘max_steps‘ is a positive integer step budget, larger for tasks needing planning,
     checking, or repair.
```

At inference time, the API call uses deterministic decoding and requests a JSON object. The returned topology is parsed into the runtime schema, selected roles are normalized against the available role set, invalid entry or manager roles are discarded, and the returned maximum step count is clamped to the configured budget.

C.2 Agent Execution Prompts

During the online execution phase, we employ a role-conditioned prompt design. The orchestrator dynamically injects distinct schemas and instructions based on the currently assigned role, enabling highly differentiated outputs (e.g., Planner, Builder, Checker, Refactor) within a unified execution framework. The LLM remains the primary generator for plans, code, and answer candidates, while deterministic local tools are invoked selectively for structured analysis, verification, execution, and formatting.

C.2-A Role-Conditioned Execution Prompt

```
[System]
You are the {role_key} role in a multi-agent system.
Return ONLY a JSON object that matches the required schema exactly.
Do not include markdown or extra keys.
```

```
[User]
Task: {task_text}
Agent name: {agent_name}
Agent domain_tags: {domain_tags}
Agent role_tags: {role_tags}
Agent tool_tags: {tool_tags}
Constraints: {constraints_json}
Upstream outputs: {upstream_json}
Available roles: {available_roles_json}
Available tools: {available_tools_json}
Tool history: {tool_history_json}

Schema:
- ... role-specific fields ...

Guidance:
- ... role-specific instructions ...
```

C.3 Decoding Settings

Temperatures are assigned per function family a priori and were not tuned on test outcomes (Table 8). Selection and deterministic local tools involve no LLM call.

Table 8: Decoding temperature by stage and function family.

Stage	**Function family**	**Temperature**
Topology planning	JSON-schema topology decision	0.0 (deterministic)
Selection / search	LLM-free MCTS	N/A (no LLM call)
Deterministic local tools	parser / executor / test runner	N/A (no LLM call)
LLM-assisted deterministic functions	verify / normalize / format	0.1
Standard construction and reasoning	plan / solve / code generation	0.2
Exploratory functions	research / brainstorm / hypothesis generation	0.6
Offline ecosystem synthesis (frozen before evaluation)	tool / theme / profile generation	0.5

D Protocol for Harder and Open-Ended Benchmarks

This appendix gives the protocol behind Table 2.

Shared protocol. Evaluation subsets are pre-registered with a fixed seed before any model run. Gold labels and hidden tests are visible only to the offline evaluator. No evaluator-feedback repair loops, external retrieval, or knowledge APIs are used. All methods share one pinned backbone per suite. The GPT-4o router and SMART select from the identical frozen 300-agent pool of Appendix A; no suite-specific agents or tools are added, and all reward-side and search constants are those of Table 9.

HotpotQA (distractor). 500 questions from the official dev split with the official 10 candidate paragraphs per question; backbone gpt-4o-mini for every method. All methods use the same five QA tool function families: context indexing, question decomposition, evidence-chain construction, supporting-fact verification, and answer extraction/normalization. Answer F1 scores the final answer; Joint F1 jointly scores the answer and the identified supporting facts. The gap vector reads only the team's own intermediate outputs (sub-questions resolved, evidence collected, facts verified, answer extracted); gold supporting facts remain hidden from inference.

BigCodeBench-Instruct. 200 tasks from the Instruct split; backbone gpt-4o-mini; hidden tests are used only by the evaluator.

Humanity's Last Exam. 300 questions chosen from the 976 text-only Math questions of the official test split; only the question text enters inference. Because the suite lies beyond the discriminative band of gpt-4o-mini, all methods use gpt-5-mini with `reasoning_effort=medium`; the router is a GPT-4o selector over gpt-5-mini executors.

Scope. The mechanism requires only that execution exposes some checkable notion of remaining work, not crisp success criteria at run time. Tasks that expose no checkable intermediate state (e.g., free-form creative work) deactivate the Gap term; the mean-active aggregation then falls back to Unary+Trans+IO, so team selection still functions but without a progress signal. Since success itself cannot then be verified, evaluation is ill-defined for any method, and we treat truly subjective tasks as out of scope.

E Detailed Case Study and Visualization Analysis

To intuitively demonstrate the effectiveness of our dynamic orchestration and LLM-free MCTS routing, we provide a detailed analysis of a complex code generation task.

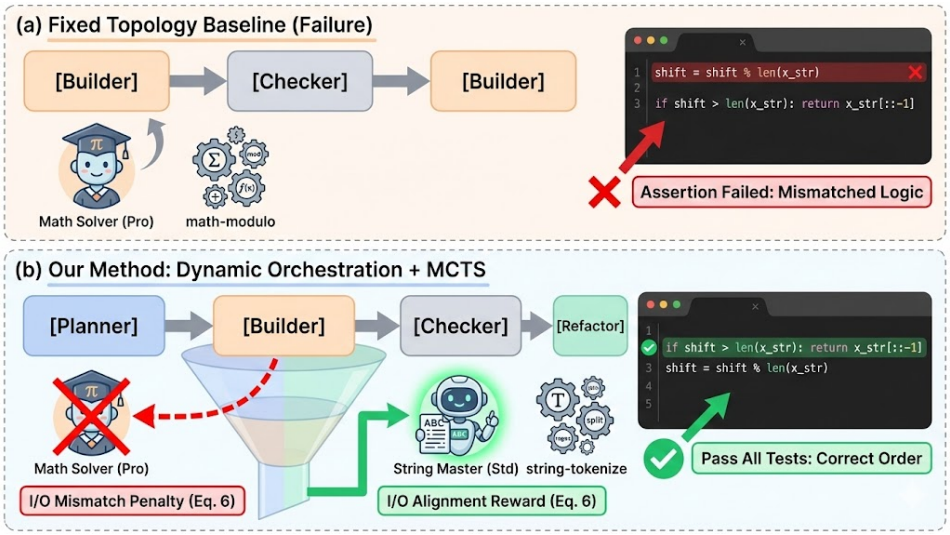


Figure 4: Comparison of execution trajectories between the fixed topology baseline and our dynamically orchestrated MCTS framework on the `HumanEval_65_circular_shift` task. MCTS successfully avoids a capability-mismatched generalist agent and selects a highly compatible string specialist.

Case Selection and Motivation. We select `HumanEval_65_circular_shift` as a representative challenging case. The task requires strict rule compliance: if `shift` is larger than the digit length, return reversed digits. This serves as an excellent stress test to evaluate whether the system can choose an agent with the correct I/O behavior and specific domain expertise rather than blindly defaulting to agents with higher general model capacity.

Fixed Topology Baseline (Figure 4a). In standard fixed-pipeline approaches (e.g., MetaGPT-style architectures), the execution graph is hardcoded as `Builder` $\rightarrow$ `Checker` $\rightarrow$ `Builder`. On this task, the baseline deterministically assigns `MathematicalSolver-HighlyPrecise-Pro` as the Builder. Although this agent possesses high general capabilities, its tool trajectory is inherently dominated by math-oriented operators (e.g., `math-number-theory`, `math-modulo`). Consequently, it repeatedly attempts to rewrite the code around modular arithmetic. This leads to a subtle specification violation: the agent performs `shift = shift % len(x_str)` before checking if `shift > len(x_str)`, rendering the required reversal branch completely ineffective for edge cases like `(97, 8)`. As a result, the baseline ultimately fails the assertion `candidate(97, 8) == '79'`.

Our Method: Dynamic Orchestration + MCTS (Figure 4b). In contrast, our dynamic planner first expands the workflow architecture to a more suitable configuration: `{Planner, Builder, Checker, Refactor}`. During the online role-filling phase, MCTS searches through the massive pool and assigns:

- `Planner`: `AlgorithmArchitect-HighlyPrecise-Pro`
- `Builder`: `StringMaster-Moderate-Std`

Notably, the selected Builder leverages string-oriented tools (e.g., `string-parsing-tokenize`) and successfully preserves the required conditional branch order (`if shift > len(x_str)` before applying modulo normalization), passing all test cases. This demonstrates that dynamic role decomposition combined with MCTS can recover the correct implementation path even when fixed reasoning chains drift into inappropriate tool families.

Candidate-level Analysis and the Advantage of Eq. (6). For the critical implementation role (Builder), MCTS navigates a large candidate reservoir (66 Builder-capable agents out of 300). The core competition occurs between the generalist `MathematicalSolver-HighlyPrecise-Pro` and the specialist `StringMaster-Moderate-Std`. While standard routing methods might greedily pick the former due to its higher base capability score, our MCTS utility function explicitly incorporates inter-agent I/O alignment and artifact-gap reduction.

Because the generalist's tool profile and output modality exhibit a higher mismatch with the upstream Planner's context for this specific string-manipulation task, and because the unresolved code artifact gaps favor string-oriented construction, it is penalized relative to the specialist. Consequently, the `StringMaster` agent receives stronger I/O and gap-reduction components, leading MCTS to select it. This highlights the core advantage of our framework in massive candidate pools: the ability to discover and route to the most cost-effective, task-aligned specialist, rather than blindly relying on the strongest generic model.

F Experimental Details and Hyperparameters

Table 9 summarizes the code-level parameters.

Table 9: Key hyperparameters used by the current SMART routing and MCTS evaluation pipeline. Long implementation flag names are shortened for readability; the standalone retrieval API has service defaults `top_n=50` and `top_k=10`, while the table reports paper-mode benchmark defaults.

Component	Symbol / short name	Description	Value
Retrieval & MMR	λ	Relevance-diversity trade-off in MMR	0.50
	N	Initial ANN retrieval pool in paper mode	30
	K	MMR-retained candidates per role in paper mode	15
	K_f	Candidate cap used by MCTS	5
Unary scoring	μ	Cost penalty coefficient	0.05
	ν	Latency penalty coefficient	0.05
	w_m	Model-tier match bonus	0.08
	w_e	Function-band match bonus	0.06
MCTS search	reward	Active reward combiner	mean active
	Trans on	Include transition posterior component	true
	IO on	Include adjacent I/O compatibility when available	true
	Gap on	Include artifact-gap reduction component	true
	C_{ucb}	UCB exploration constant	1.414
	H	Maximum rollout depth	4
	B	MCTS iteration budget	64
Reward details	γ	Discount factor for rollout return	0.95
	I/O aggregation	Unweighted mean of format, modality, and tool-tag overlaps	$(F + M + T)/3$
	gap aggregation	Equal mean over active artifact gaps	uniform
	gap threshold	Score threshold for simulated next-gap closure	0.55
Benchmark protocol	cache	Persistent bandit cache in evaluation scripts	false
	x-inst. update	Cross-instance posterior updates in evaluation scripts	false

Parameter Selection Strategy. We set $\lambda = 0.5$ to balance semantic relevance and diversity in MMR, and keep unary cost and latency penalties small so that high-capability agents are not eliminated solely by metadata cost tiers. MCTS uses the standard $C_{\mathrm{ucb}} \approx \sqrt{2}$ and $\gamma = 0.95$. The default `mean_active` reward averages the enabled normalized components in Eq. (6).

How the defaults were chosen. The defaults were set by scale calibration, not test-set optimization, and fixed once before any benchmark evaluation: semantic relevance remains the dominant unary signal, w_m and w_e provide smaller tie-breaking adjustments for tier/band compatibility, and the

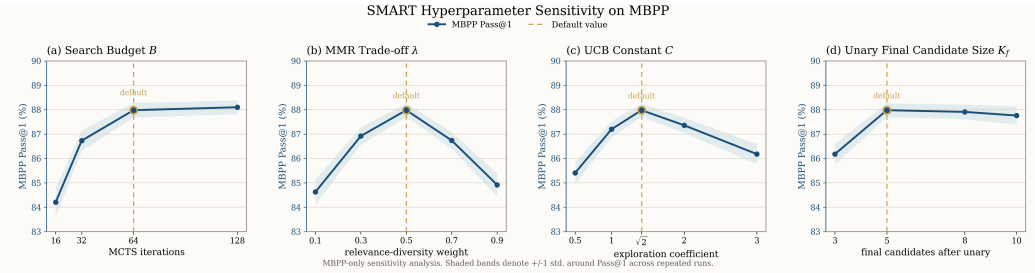


Figure 5: Hyperparameter sensitivity of SMART on MBPP.

symmetric $\mu = \nu$ discourage expensive or slow agents without letting cost tiers dominate suitability. The search defaults ($\lambda = 0.5$, $C_{\text{ucb}} = \sqrt{2}$, $\gamma = 0.95$) are standard. The same constants are used unchanged across all four main benchmarks and the three suites of Section 4.3.2, the last under a different backbone, with no per-domain retuning. The sweeps in Appendix F.1 are post-hoc robustness checks and were not used to revise the defaults.

F.1 Parameter Sensitivity

We analyze the sensitivity of SMART on MBPP with respect to four key hyperparameters: the MCTS search budget B, the MMR trade-off λ, the UCB constant C, and the Unary final candidate size K_f. Since SMART uses LLM-free rollouts and these sweeps do not affect API cost, we report MBPP Pass@1 only in Figure 5. The curves should be interpreted as a default-selection check rather than evidence that SMART is invariant to every setting. Performance improves clearly as B increases to 64 and then saturates. For λ and C, the sweep shows a moderate-preference band: $\lambda = 0.5$ and $C \approx \sqrt{2}$ are near the observed best settings, while extreme relevance-only/diversity-heavy retrieval or too little/too much UCB exploration performs worse. Increasing K_f from 3 to 5 helps noticeably, while larger values dilute the fixed search budget. These results support the chosen defaults, but we do not claim broad insensitivity to λ or C.

Reward-side constants. Table 10 sweeps the reward-side constants one at a time around the defaults on the full MBPP test split (gpt-4o-mini). Cost is the total API cost over the split (accounting of Figure 2a); the default setting reproduces the original run (87.68 at \$0.619). Removing either bonus (w_m or w_e) costs 2.3–2.9 pp and raises cost, because mismatched selections trigger retries; doubling either has little effect. $\mu = \nu$ acts as an explicit accuracy–cost dial rather than a fragile hyperparameter: within $[0, 0.10]$ accuracy varies by at most 1.5 pp while cost falls monotonically, and only the $6\times$ penalty (0.30) degrades accuracy visibly. Halving or doubling any default moves Pass@1 by about 1 pp or less; visible degradation requires removing a bonus entirely or the $6\times$ penalty, so the defaults sit on a plateau rather than a knife-edge. This flatness is structural: min–max normalization over the shortlist (Eq. 2) cancels absolute scales, selection is a ranking decision that changes only when a perturbation flips the shortlist order, and the mean-active aggregation (Eq. 6) keeps any single coefficient from dominating. We claim a stable operating region, not global insensitivity.

Table 10: One-at-a-time sweep of reward-side constants on MBPP (gpt-4o-mini): Pass@1 (%) / total API cost (\$). Default values are marked with †.

$\mu = \nu$	0 88.86 / 0.798	0.05† 87.68 / 0.619	0.10 87.39 / 0.607	0.30 79.77 / 0.562
w_m	0 85.34 / 0.652	0.04 87.10 / 0.623	0.08† 87.68 / 0.619	0.16 86.80 / 0.631
w_e	0 84.75 / 0.637	0.03 86.80 / 0.623	0.06† 87.68 / 0.619	0.12 87.97 / 0.625

F.2 Run-to-run Variability and Significance

LLM execution, task sampling, candidate retrieval, and MCTS search can all introduce variance. To quantify it, we re-ran SMART (both settings) and MasRouter five independent times each on every

benchmark. EvoFlow could not be re-run because its public implementation is no longer available, so its entries are the values reported in its original paper, as in Table 1. Table 11 reports mean $\pm$ standard deviation (1-sigma) over the five runs.

Table 11: Mean $\pm$ SD over five independent runs. EvoFlow is the value reported in its paper.

Benchmark	**SMART (gpt-4o-mini)**	**EvoFlow (gpt-4o-mini)**	**SMART (LLM-Pool)**	**MasRouter (LLM-Pool)**
GSM8K	93.67 ± 0.18	92.90	94.90 ± 0.26	93.69 ± 0.21
MATH (L5)	61.85 ± 0.56	57.70	62.55 ± 0.50	56.42 ± 0.59
HumanEval	94.35 ± 1.02	92.85	94.96 ± 1.39	90.08 ± 1.21
MBPP	87.80 ± 0.74	84.50	88.91 ± 0.76	83.93 ± 0.76

Rerun means match the reported values (SMART within 0.46 pp of Table 1; our MasRouter reruns within 0.54 pp of its published numbers), so the reported numbers are not favorable draws, and SDs are small throughout. Every SMART (LLM-Pool) run exceeds every MasRouter run on all four benchmarks (exact two-sided Mann–Whitney U test, $p = 0.008$ per benchmark; mean margins $+1.21$/$+6.13$/$+4.88$/$+4.98$ pp on GSM8K/MATH/HumanEval/MBPP). All five SMART (gpt-4o-mini) runs exceed EvoFlow's reported value on every benchmark (one-sided sign test, $p = 0.031$; mean margins $+0.77$/$+4.15$/$+1.50$/$+3.30$ pp; worst-run margins $+3.41$/$+2.60$ pp on MATH/MBPP).

F.3 Latency Decomposition versus Pool Size

Search budgets follow a fixed sub-linear schedule of the pool size P (Table 12); the $P = 300$ row is the default configuration of Table 9. Table 13 decomposes per-step selector overhead and per-task wall-clock time on the full MBPP test split. Three observations follow. (i) Total selector overhead per step (ANN+MMR, unary scoring, and MCTS) grows from 66 to 138 ms, about $2\times$ for a $30\times$ larger pool, and the $P = 300$ sum reproduces the 138 ms/step selector latency of Table 4; MCTS time tracks the iteration budget at roughly 1.9 ms per iteration independent of pool size, because rollouts operate on the K_f-capped candidate set at fixed depth. (ii) End-to-end latency decreases from 136.0 to 102.0 s per task as the pool grows: richer pools yield better specialist matches and fewer LLM retries (LLM API time 116.5 to 86.0 s per task), while selector overhead remains a small fraction of end-to-end time throughout, mirroring Figure 3. (iii) Because the schedule keeps the per-candidate search intensity roughly constant ($B/K_f \approx 10$–13 iterations per candidate), the accuracy gain with pool size reflects pool richness rather than an enlarged search budget.

Table 12: Search-budget schedule as a function of pool size P.

P	ANN top-N	MMR K	MCTS K_f	MCTS iterations B	Rollout depth
10	10	6	3	32	4
30	15	8	4	40	4
100	22	12	5	52	4
300	30	15	5	64	4

Table 13: Per-step selector overhead (ms) and per-task time (s) on the full MBPP test split as the pool grows (gpt-4o-mini).

P	ANN+MMR (ms/step)	Unary (ms/step)	MCTS (ms/step)	LLM API (s/task)	Tool exec (s/task)	End-to-end (s/task)	Pass@1 (%)
10	4	2	60	116.5	19.3	136.0	71.55
30	6	3	76	101.0	17.8	119.0	78.30
100	9	4	96	92.0	16.8	109.0	84.75
300	13	6	119	86.0	15.8	102.0	87.68

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction scope SMART to controlled role-conditioned teaming over a 300-agent pool, and every claim (accuracy, cost, harder suites, stability across runs) is supported by Sections 4–5 and the appendices.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 5 discusses the synthetic 300-agent ecosystem, the absence of live third-party services, the requirement of checkable intermediate state (subjective tasks are out of scope), and irreversible retrieval truncation.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not present formal theorems or proofs; equations define the algorithm and scoring rules.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We provide the full prompts, the ecosystem construction process, the algorithm details, all hyperparameters including decoding temperatures (Appendix C), and the evaluation protocol of every benchmark (Section 4, Appendix D).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in

some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Code and the synthetic agent ecosystem are released at the URL given in the abstract, with instructions to reproduce the main results.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4 and Appendices A, C, D, and F specify the benchmarks, baselines, LLM backbones, decoding temperatures, prompts, agent construction, hyperparameters, and how the defaults were chosen.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Appendix F reports mean $\pm$ 1-sigma standard deviation over five independent runs for SMART and MasRouter on all four main benchmarks, with exact Mann–Whitney and sign tests; Table 1 notes the run-to-run variability. EvoFlow could not be re-run because its public implementation is no longer available.

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All experiments run through LLM APIs; we report API cost per task (Figure 2, Table 2) and per-step selector and per-task end-to-end latency (Table 4, Appendix F). LLM-free search runs on a single CPU.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The authors reviewed the NeurIPS Code of Ethics; the study uses public benchmarks and synthetic agents/tools without human subjects, personal data, or deployment.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [N/A]

Justification: This is algorithm-oriented research on team formation for multi-agent systems without a direct path to a specific application or deployment.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release high-risk pretrained models, scraped datasets, or generative systems requiring special release safeguards.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Existing benchmarks, baselines, and model providers are cited, and their public licenses and terms of use are respected.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: TODO: if the synthetic agent/tool ecosystem is released with the code, change to Yes and describe the documentation and license.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The work does not involve crowdsourcing, participant studies, or human-subject data collection.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The work does not involve human subjects, so IRB or equivalent approval is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: The paper describes the LLMs used for planning, execution, the LLM pool, and offline tool synthesis, while explicitly stating that MCTS rollouts are LLM-free.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.