FRAME: FAILURE-AWARE MULTI-AGENT TOPOLOGY LEARNING WITH RANDOM FOURIER FEATURES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-based multi-agent systems (MAS) increasingly rely on learned collaboration topologies to coordinate specialized agents efficiently. Yet, existing topology optimizers remain largely *task-conditioned*: they construct topology from the task and predefined role capabilities, while overlooking evidence revealed during execution and providing limited mechanisms to recover when the resulting collaboration fails. To mitigate this drawbacks, we introduce FRAME (**F**ailu**r**e-**A**ware **M**ulti-Agent Topology Learning with Random Fourier Fe**a**tures), *a failure-aware, execution-conditioned framework that unifies topology construction and repair through bilevel optimization*. At the upper level, FRAME learns the collaboration graph autoregressively, conditioning each decision on the task, selected roles, and evolving execution states. Specifically, FRAME introduces Random Fourier Feature Upper Confidence Bound (RFF-UCB) role selection, combining nonlinear utility estimation with uncertainty-aware exploration, while communication edges are learned using a lightweight neural network optimized directly from task reward. When performance falls below a prescribed threshold, the lower level localizes the source of underperformance and performs targeted role repair, which is subsequently fed back into topology construction. This design turns topology learning from a one-shot graph prediction problem into a closed-loop process of *constructing, executing, diagnosing, and repairing*. Across diverse benchmarks, FRAME consistently outperforms the existing topology design algorithms while being token efficient.

1 INTRODUCTION

Large language model (LLM)-based multi-agent systems (MAS) have emerged as a promising paradigm for solving complex reasoning and decision-making tasks by decomposing problems across specialized agents, exchanging reasoning, and iteratively refining solutions (Yao et al., 2022; Wu et al., 2023; Hong et al., 2024). A key determinant of their effectiveness is the underlying *collaboration topology*: the structure governing which agents participate and how information is exchanged. Early approaches relied on manually designed structures such as chains, trees, and fully connected graphs (Wei et al., 2022; Yao et al., 2023; Du et al., 2023), applying the communication irrespective of task complexity. Subsequent learning-based approaches introduced task adaptivity by pruning, reweighting, or activating agents and communication links within a predefined topology or agent pool (Zhuge et al., 2024; Zhang et al., 2024; 2025a; Wang et al., 2025). Approaches such as `AgentConductor` (Wang et al., 2026a) incorporates feedback, and commits to each turns topology before execution where topology changes happens after complete graph execution. More recently, generative approaches such as `ARG-Designer` (Li et al., 2026a) and `OFA-MAS` (Li et al., 2026b) determine team composition and communication structure in an autoregressive setting.

Despite recent progress, existing topology design methods remain largely *task-conditioned* rather than *execution-conditioned* within evolving graph leading to four key limitations. **(i) Offline topology commitment:** the collaboration graph is typically constructed from the task, candidate roles, and partial topology before agent execution begins, preventing intermediate outputs from influencing subsequent team composition. **(ii) Limited exploration:** role selection is commonly driven by similarity scores, without explicitly modeling uncertainty and therefore lack a principled mechanism for balancing exploitation with exploration of potentially useful agents. **(iii) Costly topology learning:** autoregressive generators often rely on backbone-specific collections of successful task–

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

graph trajectories and comparatively heavy learned generators, introducing substantial cold-start cost. **(iv) Failure-unaware adaptation:** when topology underperforms, existing methods provide limited support for attributing the failure to a specific component and selectively repairing it, leaving topology construction and failure recovery largely disconnected. Together, these limitations expose a fundamental mismatch between current topology optimization and the sequential nature of multi-agent collaboration: team formation should evolve with execution, using intermediate evidence to decide *who to add*, uncertainty to decide *what to explore*, and failure signals to decide *what to repair*.

To address these limitations, we introduce FRAME, a unified framework that couples execution-conditioned topology construction with targeted role repair. FRAME formulates adaptive topology learning as a conditioned bilevel optimization problem. At the upper level, the collaboration topology is constructed autoregressively while conditioning on both the task and the evolving agent states, optimizing task utility while controlling agent participation. When the performance falls below a prescribed threshold, the lower-level optimization is activated to localize the source of underperformance and selectively revise the responsible role, rather than reconstructing the entire team. For an efficient online construction, FRAME replaces a heavy parametric role generator with an uncertainty-aware RFF-UCB policy: RFFs (Rahimi & Recht, 2007) provide a compact nonlinear representation of role utility, while UCB-style exploration (Abbasi-Yadkori et al., 2011) balances predicted utility against uncertainty in next role selection. Given the selected agent, its incoming communication edges are sampled through a lightweight policy trained directly from task-level reward using REINFORCE (Williams, 1992). In summary, FRAME recasts topology design as a closed-loop learning problem, where *execution shapes team composition, uncertainty governs exploration, and failure feedback enables targeted structural and role adaptation*. We summarize our contributions as follows:

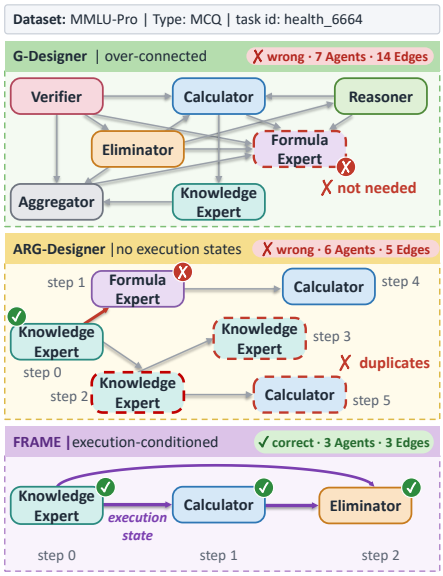


Figure 1: Comparison of the final learned communication topologies generated by `G-Designer`, `ARG-Designer`, and FRAME (Ours) for an MMLU-Pro task.

1. **Failure-aware bilevel topology optimization.** We formulate adaptive MAS design as a bilevel problem that couples *topology construction* with *failure recovery*: the upper level optimizes task utility under agent and communication costs, while the lower level is activated upon underperformance to identify and repair deficient roles.
2. **Execution-conditioned autoregressive team construction.** Unlike methods that commit to a collaboration graph before execution, FRAME interleaves topology construction with agent execution. Each partial topology is executed online, and the resulting agent states and outputs condition subsequent role-selection and communication decisions.
3. **Uncertainty-aware, parameter-efficient online topology learning.** We introduce an RFF-UCB policy for role selection that captures nonlinear role utility while explicitly balancing exploitation and uncertainty-driven exploration, together with a lightweight Bernoulli edge policy optimized by REINFORCE approach.
4. **Targeted failure localization and minimum-intervention role repair.** When collaboration underperforms, FRAME localizes the responsible role through interventional re-execution and repairs it through a minimal semantic role perturbation, preserving the remaining team structure whenever possible.

We evaluate FRAME across a wide range of benchmarks and observe consistent improvements over state-of-the-art baselines while maintaining substantially lower token consumption. Notably, the gains are most pronounced on challenging instances where the initially constructed topology is more likely to fail, highlighting the effectiveness of failure-aware topology adaptation and targeted repair.

2 Related Work

Topology Optimization for MAS. Prior works on topology design for MAS can be broadly organized into three categories based on how the collaboration structure is constructed and adapted. **(i) Predefined topologies** rely on fixed interaction protocols that remain unchanged across tasks, as exemplified by `LLM-Debate` (Du et al., 2023) and `LLM-Blender` (Jiang et al., 2023). **(ii) Learning-based topology optimization** adapts agent participation or communication patterns to task characteristics within a predefined structural or search space. Representative approaches include `GPTSwarm` (Zhuge et al., 2024), `AgentPrune` (Zhang et al., 2025a), `AgentDropout` (Wang et al., 2025), `G-Designer` (Zhang et al., 2024), and `DyLAN` (Liu et al., 2023). Recent approaches such as `ADAS` (Hu et al., 2025), `AFlow` (Zhang et al., 2025b), and `MASPOB` (Hong et al., 2026) extend this direction by jointly optimizing coordination strategies and prompts. Whereas approaches such as `FlowReasoner` (Gao et al., 2025) and `AgentConductor` (Wang et al., 2026a), incorporate feedback to refine workflows or collaboration structures. However, the feedback is incorporated after complete graph execution. In contrast, FRAME interleaves incremental topology construction with agent execution, using intermediate outputs to condition subsequent role-selection and edge-placement within the evolving graph. **(iii) Autoregressive topology construction** `ARG-Designer` (Li et al., 2026a) sequentially generates collaboration structures through autoregressive factorization, while `OFA-MAS` (Li et al., 2026b) employs a mixture-of-experts formulation and `GTD` (Jiang et al., 2026) adopts guided graph diffusion. Despite this progress, existing methods treat topology optimization and failure recovery as separate problems. Our formulation instead couples *execution-conditioned topology construction* with *performance-triggered role repair*.

Failure attribution and runtime repair in MAS. A complementary line of work studies failure attribution and recovery in MAS. Representative approaches include `Who&When` (Zhang et al., 2025c): identifies the responsible agent and decisive error step, `AgenTracer` (Zhang et al., 2026) and `FAMAS` (Ge et al., 2026) localize failures through counterfactual replay, fault injection, and repeated-execution analysis. Whereas, `DoVer` (Ma et al., 2026) validates hypothesized causes through targeted interventions and downstream re-execution, `AgentDropoutV2` (Wang et al., 2026b) performs test-time rectify-or-reject recovery. However, these methods operate after the collaboration topology is fixed. In contrast, FRAME couples failure signals back to topology construction, allowing recovery information to influence subsequent structural decisions.

3 Preliminaries

Bilevel Optimization. Bilevel optimization is a nested optimization problem with two levels, where upper level of optimization requires an optimal response from a lower-level optimization (Colson et al., 2007; Sinha et al., 2017). In particular with upper, lower level objectives as F, f and $\mathcal{X}$, $\mathcal{Y}(x)$ as upper and conditioned lower level feasible sets, the bilevel optimization solves

$$\max_{x\in\mathcal{X}} F\left(x, y^*(x)\right), \qquad \text{s.t.} \quad y^*(x) \in \arg\min_{y\in\mathcal{Y}(x)} f(x,y), \tag{1}$$

Here $y^*(x)$ denotes the optimal lower-level response conditioned on x.

Gaussian Process and Random Fourier Features. A Gaussian process (`GP`) (Williams & Rasmussen, 2006) defines a distribution over functions such that every finite collection of function values is jointly Gaussian. We denote `GP` prior as $f \sim \mathcal{GP}(m, k_\ell)$, where m and k_ℓ denote the mean and covariance functions, respectively. For inputs $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^p$, we consider $k_\ell(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x}-\mathbf{x}'\|_2^2}{2\ell^2}\right)$, $\ell > 0$ is kernel bandwidth.

Theorem 3.1 (Bochner's theorem and RFF approximation) *(Rahimi & Recht, 2007) For a continuous, shift-invariant and normalized positive-definite k_l with a spectral probability measure, the kernel approximation through RFFs $\mathbf{\Phi}(\mathbf{x})$ is given as*

$$k_\ell(\mathbf{x}, \mathbf{x}') \approx \mathbf{\Phi}^\top(\mathbf{x})\mathbf{\Phi}(\mathbf{x}'), \quad \textit{where } \mathbf{\Phi}(\mathbf{x}) = \sqrt{\frac{2}{D}}[\cos(\omega_1^\top\mathbf{x}+b_1), \ldots, \cos(\omega_D^\top\mathbf{x}+b_D)]^T \in \mathbb{R}^D, \tag{2}$$

where $\omega_r \sim \mathcal{N}(0, \ell^{-2}I_p)$ and $b_r \sim \mathrm{Unif}(0, 2\pi)$, for $r = 1, \ldots, D$.

Conditioned on the sampled features, the model $f_D(\mathbf{x}) = m(\mathbf{x}) + \mathbf{\Phi}^\top(\mathbf{x})\mathbf{w}$, with $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_D)$, has covariance $k_D(\mathbf{x}, \mathbf{x}') = \mathbf{\Phi}^T(\mathbf{x})\mathbf{\Phi}(\mathbf{x}')$. It therefore defines a finite-rank approximation to the original GP, rather than an exact finite-dimensional representation of it.

4 FAILURE-AWARE MULTI-AGENT TOPOLOGY LEARNING

In this section, we present the proposed framework, FRAME. We first introduce the problem setup and formulate the topology learning as a bilevel optimization problem, followed by the methodology.

Setup. We consider a MAS with N agents, denoted by $\mathcal{A} = \{a_i\}_{i=1}^N$. Each agent a_i is represented by a tuple (r_i, s_i), where r_i, s_i denote its assigned role and execution state. The interactions among a selected subset of agents are modeled using a directed acyclic graph (DAG) $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} \subseteq \mathcal{A}$ denotes the participating agents and $\mathcal{E}$ represents their directed communication links.

Problem Statement. Given a task q and an agent pool $\mathcal{A}$, the goal is to autoregressively learn a task-adaptive topology $\mathcal{G}^*$ conditioning on the evolving execution states and the failure states while maximizing task utility and controlling structural cost.

Bilevel Formulation. To efficiently handle topologies in the *underperformance regime* (performance less than acceptable threshold) while jointly adapting the agent roles, states and communication structure, we formulate the topology learning problem as a bilevel optimization. This provides a natural mechanism to couple *state-conditioned autoregressive topology generation* with *failure-aware minimum-intervention role repair*.

Let $\mathcal{R} = \{r_i\}_{i=1}^N$ denote the initial role configuration and $\mathcal{S} = \{s_i\}_{i=1}^N$ the agent states after graph execution. The complete pipeline is illustrated in Figure 2 and formalized in Algorithm 1 and 2 (in Appendix C.2). At the upper level, FRAME autoregressively constructs an initial topology G, conditioned on the task q, the current roles, and the evolving execution states. If its average task reward satisfies the target threshold τ, the generated topology is retained without intervention. Otherwise, the system enters the *underperformance regime*, activating the lower-level optimization to determine the minimum modification to the roles required to recover acceptable performance. The corrected roles are subsequently fed back to the upper-level topology generator, allowing the communication structure to adapt to the identified failure. The bilevel optimization is formulated as

$$\begin{aligned} &\textit{Upper:} && \max_{G \in \mathcal{G}(q;\mathcal{R},\mathcal{S})} U\left(q, G_{\mathcal{R}^\star(G)}, \mathcal{R}^\star(G)\right), \\ \text{s.t.} \quad &\textit{Lower:} && \mathcal{R}^\star(G) \in \underset{\widetilde{\mathcal{R}} \in \Omega_{\mathcal{R}}(G)}{\arg\min}\ \Delta_{\text{sem}}\left(\widetilde{\mathcal{R}}, \mathcal{R}\right) \\ & && \qquad \text{s.t.} \quad z(G)\left[\tau - \bar{J}\left(q, G_{\widetilde{\mathcal{R}}}, \widetilde{\mathcal{R}}\right)\right] \le 0, \\ & && z(G) = \mathbb{I}\left[\bar{J}(q, G, \mathcal{R}) < \tau\right]. \end{aligned} \tag{3}$$

Here, $\bar{J}(q, G, \mathcal{R})$ denotes the average task reward with graph G and initial roles, τ is the target performance. Here $z(G)$ is an indicator variable activated when $\bar{J}(q, G, \mathcal{R}) < \tau$. Under low performance regime i.e, lower level seeks the minimum semantic role intervention such that regenerated topology achieves the target performance τ using perturbed roles. An unsuccessful candidate search is reported as repair failure, rather than solution to lower level problem. Details on semantic cost $\Delta_{\text{sem}}(\widetilde{\mathcal{R}}, \mathcal{R})$ and feasible role-repair $\Omega_{\mathcal{R}}$ are discussed in Section 4.2. In what follows, we define the upper level objective function U based on autoregressive formulation.

4.1 STATE-AWARE AUTOREGRESSIVE TOPOLOGY LEARNING

We now describe the upper-level optimization that constructs a topology $G \in \mathcal{G}(q; \mathcal{R}, \mathcal{S})$. Rather than generating the complete topology at once, FRAME constructs it autoregressively through two decisions at each step: *(i) role selection*, which selects the next agent from the available role pool, and *(ii) edge placement*, to determines the incoming communication links from the previously selected agents to the newly selected agent. After each step, the resulting partial topology is executed, and its updated execution state is used to condition the subsequent decisions.

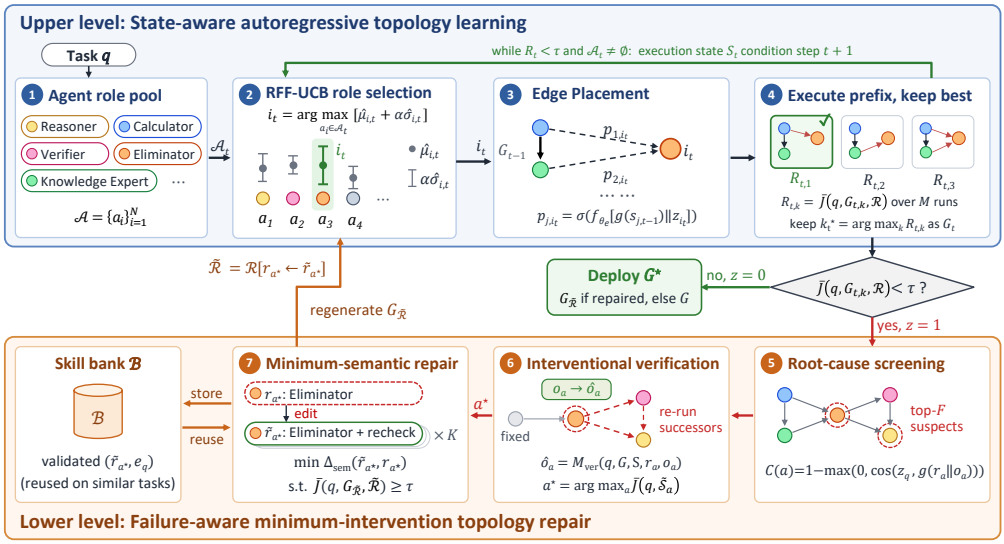


Figure 2: **Overview of FRAME. Upper-Level** autoregressively constructs the topology, selecting roles and edges conditioned on execution state. **Lower-Level** activates when performance falls below threshold, localizing the responsible agent and applying a minimum-semantic role repair.

Let $G_{t-1} = (\mathcal{V}_{t-1}, \mathcal{E}_{t-1})$ denote the partial topology after $t-1$ construction steps, $\mathcal{S}_{t-1}$ its corresponding execution state, and the available agents are $\mathcal{A}_t = \mathcal{A} \setminus V_{t-1}$, with indexed candidate roles $\mathcal{R}_t = (r_i)_{a_i \in \mathcal{A}_t}$. We define $h_t = (q, G_{t-1}, S_{t-1}, \mathcal{R}_t)$, $\mathbf{E}_{\text{in},t} = (e_{j,i_t})_{a_j \in V_{t-1}} \in \{0,1\}^{|V_{t-1}|}$.

The node-and-edge policy factorizes as

$$\pi(i, \mathbf{E}_{\text{in},t} \mid h_t) = \underbrace{\pi_{\Psi,\mathbf{y}}(i \mid h_t)}_{\textit{Role Selection}} \times \underbrace{\pi_{\theta_e}(\mathbf{E}_{\text{in},t} \mid h_t, i)}_{\textit{Edge Placement}}. \tag{4}$$

Importantly observe from equation 4 both structural decisions are conditioned on the state of the current graph prefix. In what follows, we define the RFF−UCB role-selection policy $\pi_{\Psi,\mathbf{y}}$ and then the stochastic edge policy π_{θ_e}.

Role Selection via RFF-UCB. We discuss role-selection policy $\pi_{\Psi,\mathbf{y}}$ introduced in equation 4. To do so, at step t, we represent each candidate agent $a_i \in \mathcal{A}_t$ with $\mathcal{A}_t = \mathcal{A} \setminus \mathcal{V}_{t-1}$ through a context vector then leverage this to construct the node selection criterion. We define context vector as

$$\mathbf{x}_{i,t} = [c_{i,t}, d_{i,t}]^T \in \mathbb{R}^2, \tag{5}$$

where $c_{i,t}$ captures the contextual relevance of agent a_i, $d_{i,t}$ measures its diversity with respect to the selected agents. Let g be the text-embedding function and define $\mathbf{z}_q = g(q)$, $\mathbf{z}_i = g(r_i)$, and $\mathbf{z}_{S_{t-1}} = g(o_{t-1}^{\text{acc}})$, where o_{t-1}^{acc} is the answer retained from the accepted graph prefix. We define

$$c_{i,t} = \eta \cos(\mathbf{z}_q, \mathbf{z}_i) + (1-\eta) \cos(\mathbf{z}_{\mathcal{S}_{t-1}}, \mathbf{z}_i), \qquad d_{i,t} = 1 - \max_{a_j \in \mathcal{V}_{t-1}} \cos(\mathbf{z}_i, \mathbf{z}_j), \tag{6}$$

where $\eta \in [0,1]$ controls the trade-off between task-level relevance and execution-conditioned relevance. Thus, $c_{i,t}$ favors agents whose roles are aligned with both the original task and the information produced by the current graph prefix, while $d_{i,t}$ favors agents whose roles provide complementary information relative to the agents already selected.

At the first construction step, we set $c_{i,1} = \cos(\mathbf{z}_q, \mathbf{z}_i)$ computed solely from task to role similarity. Similarly, since $\mathcal{V}_0 = \varnothing$ we have $d_{i,1} = 1$. The resulting context $\mathbf{x}_{i,t}$ then evolves with the execution of the partial topology. To model the expected utility as a nonlinear map on these contextual features, we approximate the context-dependent shaped node reward using a shared nonlinear predictor leveraging Theorem 3.1,

$$\mu(\mathbf{x}) \approx \mathbf{\Phi}(\mathbf{x})^\top \mathbf{w}, \qquad \mathbf{w} \in \mathbb{R}^D, \tag{7}$$

where $\mathbf{\Phi}(x)$ is RFF features with inputs as relevance-diversity context rather than a role-specific parameter vector. Following the LinUCB (Abbasi-Yadkori et al., 2011) design, we maintain the

ridge-regression statistics $\mathbf{\Psi} \in \mathbb{R}^{D\times D}$ and $\mathbf{y} \in \mathbb{R}^D$, initialized once at training start as $\mathbf{\Psi} = \lambda \mathbf{I}_D$ and $\mathbf{y} = \mathbf{0}$, with $\lambda > 0$. Then for each available candidate $a_i \in \mathcal{A}_t$, weight, predicted reward and exploration scale are estimated as

$$\widehat{\mathbf{w}} = \mathbf{\Psi}^{-1}\mathbf{y}, \qquad \widehat{\mu}_{i,t} = \mathbf{\Phi}^\top(\mathbf{x}_{i,t})\widehat{\mathbf{w}}, \qquad \widehat{\sigma}_{i,t} = \sqrt{\mathbf{\Phi}(\mathbf{x}_{i,t})^\top \mathbf{\Psi}^{-1}\mathbf{\Phi}(\mathbf{x}_{i,t})}.$$

The next agent is then selected according to the RFF−UCB criterion as

$$i_t = \underset{i:a_i\in\mathcal{A}_t}{\arg\max} \left[\widehat{\mu}_{i,t} + \alpha\widehat{\sigma}_{i,t}\right], \tag{8}$$

where $\alpha > 0$ controls the exploration-exploitation trade-off. We have $\pi_{\mathbf{\Psi},\mathbf{y}} = \mathbb{I}(i = i_t)$. Let R_{t-1} and R_t denote the empirical rewards of the accepted prefixes before and after adding a_{i_t}. We assign node addition reward as

$$\gamma_t = \frac{1}{2}\left[(R_t - R_{t-1}) + R_t\right], \qquad R_0 = 0. \tag{9}$$

Observe reward combines improvement in accepted-prefix performance with absolute post-addition performance and marginal contribution. Let $\mathcal{D}_q$ be the multiset of accepted context-reward observations collected across all K_G construction runs for training task q. The statistics are updated as

$$\mathbf{\Psi} \leftarrow \mathbf{\Psi} + \sum_{(\mathbf{x},\tau_{\text{obs}})\in\mathcal{D}_q} \mathbf{\Phi}(\mathbf{x})\mathbf{\Phi}(\mathbf{x})^\top, \qquad \mathbf{y} \leftarrow \mathbf{y} + \sum_{(\mathbf{x},\tau_{\text{obs}})\in\mathcal{D}_q} \gamma_{t,\text{obs}}\mathbf{\Phi}(\mathbf{x}). \tag{10}$$

All construction runs for the current task use the same pre-task statistics. Each accepted node addition contributes one observation; the sampled edge configurations and repeated LLM executions determine its reward. The statistics are not reinitialized between tasks and are not updated during evaluation. Thus, within-task autoregressive adaptation comes from changing contexts, while reward-model adaptation occurs between training tasks.

Edge Placement. After selecting a_{i_t}, we determine its incoming edges from previously selected agents. For $a_j \in V_{t-1}$, let $s_{j,t-1} = S_{t-1}(a_j)$ be execution state, then edge scorer assigns

$$p_{j,i_t} = \sigma\left(f_{\theta_e}\left[g(s_{j,t-1})\|z_{i_t}\right]\right), \tag{11}$$

where $\|$ denotes concatenation and $f_{\theta_e} : \mathbb{R}^{2d} \to \mathbb{R}$. Conditioned on the current prefix, incoming-edge indicators are sampled independently: $e_{j,i_t} \sim \text{Bernoulli}(p_{j,i_t})$. Hence,

$$\pi_{\theta_e}(\mathbf{E}_{\text{in},t} \mid h_t, i_t) = \prod_{a_j\in V_{t-1}} p_{j,i_t}^{e_{j,i_t}}(1 - p_{j,i_t})^{1-e_{j,i_t}}. \tag{12}$$

Further we sample K_E incoming-edge masks, where sampling matrix $\mathbf{E}_{\text{in},t}^{(k)}$ determines the candidate graph $G_{t,k} = \left(V_{t-1} \cup \{a_{i_t}\}, E_{t-1} \cup \{(a_j, a_{i_t}) : e_{j,i_t}^{(k)} = 1\}\right)$. Then, each candidate graph is evaluated through M rollouts to account for LLM stochasticity and the reward is observed as

$$R_{t,k} = \bar{J}(q, G_{t,k}, \mathcal{R}) = \frac{1}{M}\sum_{m=1}^{M} J(q, G_{t,k}, \mathcal{R}; \xi_{t,k,m}). \tag{13}$$

Further we retain the highest-mean-reward candidate: $k_t^\star \in \underset{k=1,\ldots,K_E}{\arg\max} R_{t,k}$, $G_t = G_{t,k_t^\star}$, $R_t = R_{t,k_t^\star}$. The execution record retained for this candidate defines S_t and its accepted answer o_t^{acc}, which conditions the next node-selection decision. The graph-level utility is

$$U\big(q, G, \mathcal{R}^\star(G)\big) = \bar{J}\big(q, G_{\mathcal{R}^\star(G)}, \mathcal{R}^\star(G)\big) - \lambda_E\left|\mathcal{E}_{\mathcal{R}^\star(G)}\right|, \tag{14}$$

where $\lambda_E \geq 0$ controls the edge-count cost. For now observe without intervention $\mathcal{R}^\star$ is $\mathcal{R}$ and $\bar{J}\big(q, G_{\mathcal{R}^\star(G)}, \mathcal{R}^\star(G)\big) = \bar{J}\big(q, G_{\mathcal{R}(G)}, \mathcal{R}(G)\big)$ and is the reward of topology observed before termination with the initial roles.

For each sampled mask, let $U_{t,k}$ denote the edge-training utility. To promote sparse edges, we have $U_{t,k} = R_{t,k} - \lambda_E\left\|\mathbf{E}_{\text{in},t}^{(k)}\right\|_0$, $\lambda_E \geq 0$. The choice $\lambda_E = 0$ gives raw-reward training. Let us define $l_{t,k} = \sum_{a_j\in V_{t-1}}\left[e_{j,i_t}^{(k)}\log p_{j,i_t} + (1 - e_{j,i_t}^{(k)})\log(1 - p_{j,i_t})\right]$. The edge loss at step t and total edge loss for T steps is computed as

$$\mathcal{L}_{\text{edge},t} = -\frac{1}{K_E}\sum_{k=1}^{K_E}(U_{t,k} - \beta_t)\,\ell_{t,k}, \qquad \mathcal{L}_{\text{edge}} = \sum_{t=1}^{T}\mathcal{L}_{\text{edge},t}. \tag{15}$$

β_t is the baseline for the variance reduction. Gradient optimization updates the edge policy parameters whereas node-model updates use the sufficient statistics separately.

4.2 FAILURE-AWARE MINIMUM INTERVENTION TOPOLOGY REPAIR

To handle underperformance of constructed topology, FRAME first identifies the agent responsible for the observed failure and restricts the subsequent minimum-intervention repair to its role.

Root-Cause Screening and Interventional Verification. Let $G = (\mathcal{V}, \mathcal{E})$ denote the generated topology. For, $a \in \mathcal{V}$, let $o_a = \mathcal{S}(a)$ be executed output, r_a be role and $\mathbf{z}_q$ be the task embedding. We compute an semantic divergence score for screening candidate intervention targets as

$$C(a) = 1 - \max\left(0, \cos\left(\mathbf{z}_q, g(r_a \| o_a)\right)\right), \tag{16}$$

observe larger $C(a)$ indicates greater semantic inconsistency between the task and the role-conditioned output. Based on $C(a)$ we consider top F agents as candidate failure sources.

Since semantic divergence alone does not establish causal responsibility, we further verify each candidate through targeted intervention. Following `DoVer` (Ma et al., 2026), for each suspect agent a, a verification model is assigned to generate a corrected output as

$$\widehat{o}_a = \mathcal{M}_{\text{ver}}\left(q, G, \mathcal{S}, r_a, o_a\right). \tag{17}$$

observe that the task, topology, and remaining execution states was fixed. The original output o_a is then replaced by $\widehat{o}_a$, and only the downstream successors of a are re-executed according to the fixed topology, The resulting counterfactual reward is denoted by $J_a^{\text{cf}} = \bar{J}\left(q, \widetilde{\mathcal{S}}_a\right)$. The verified failure source is then selected as the agent whose targeted intervention yields the target performance. The identified agent ($a^\star$) subsequently restricts the lower-level feasible repair space to interventions on its role $r_{a^\star}$, enabling the minimum-semantic role correction.

Minimum-Semantic Role Repair. Once the responsible agent $a^\star$ is identified, we solve the lower-level optimization in equation 3 by minimally modifying its role while keeping the remaining roles unchanged. Specifically, the objective is to find a corrected role that resolves the underperformance regime while remaining semantically close to the original role. We define cost function as

$$\Delta_{\text{sem}}\left(\widetilde{r}_{a^\star}, r_{a^\star}\right) = 1 - \cos\left(g(\widetilde{r}_{a^\star}), g(r_{a^\star})\right), \tag{18}$$

where $\widetilde{r}_{a^\star}$ denotes a candidate correction of the identified role. Accordingly, the repaired role configuration is $\widetilde{\mathcal{R}} = \mathcal{R}\left[r_{a^\star} \leftarrow \widetilde{r}_{a^\star}\right]$, such that all roles other than $r_{a^\star}$ remain unchanged. Details of candidate role generation are provided in Appendix C.3. $G_{\widetilde{\mathcal{R}}}$ denotes the topology obtained by re-running the upper-level autoregressive construction using the corrected role configuration in equation 3. Finally, we emphasize that the minimum-semantic constraint is important because optimizing role correction solely for the current task may result in task-specific role drift, thereby reducing the usefulness of the repaired role on related tasks.

For efficient reuse of successful interventions, FRAME stores the corrected role $\widetilde{r}_{(a^\star)}$ together with the corresponding task embedding $\mathbf{e}_q$ in the skill bank $\mathcal{B}$. For a new task, previously validated repairs are retrieved based on task- and role-level semantic similarity and, when a sufficiently similar match is found, the stored correction is reused instead of generating K new repair candidates.

5 NUMERICAL EXPERIMENTS

Datasets, Metrics and Baselines. We evaluate FRAME across three task categories. **(1) General Reasoning** includes MMLU and MMLU-Pro (Wang et al., 2024). **(2) Mathematical Reasoning** includes GSM8K, MultiArith, SVAMP and AQuA. **(3) Code Generation** includes HumanEval and LiveCodeBench (LCB) (Jain et al., 2025) following the prior works (Li et al., 2026a; Jiang et al., 2026). In addition to the benchmarks used in ARG-Designer (Li et al., 2026a), we further evaluate FRAME on the more challenging MMLU-Pro and LiveCodeBench benchmarks. We use accuracy as metric for all datasets except HumanEval and LiveCodeBench, where we report Pass@1. More details on dataset statistics, task-specific agent pools, corresponding role descriptions and the baselines considered are provided in Appendix D.1 and D.2.

Implementation Details. We encode task, role, and execution-state representations using `all-MiniLM-L6-v2` (Wang et al., 2020) ($d = 384$). For code-generation tasks, we use the code-aware `UniXcoder` (Guo et al., 2022) encoder. Then we form the context representation following equation 6, which are then projected to a D-dimensional RFF space. Candidate roles are then selected according to equation 8. Given the selected role, the edge policy samples $K_E = 3$ candidate communication structures, yielding multiple topology rollouts. To account for stochasticity

Table 1: Performance comparison (%) on six benchmarks. ▲ / ▼ indicate absolute gain/drop relative to that backbone's own `Vanilla` row. Bold: best in a block; underline: second best.

METHOD	MMLU	GSM8K	AQuA	MultiArith	SVAMP	HumanEval	Avg.
Closed-source backbone: `GPT-4o`							
`Vanilla`	80.4	82.3	71.1	93.1	86.6	71.4	80.8
`CoT`	81.7▲1.3	86.5▲4.2	73.6▲2.5	93.3▲0.2	87.4▲0.8	74.7▲3.3	82.8▲2.0
`SC (CoT)`	83.7▲3.3	81.6▼0.7	75.6▲4.5	94.1▲1.0	88.6▲2.0	79.8▲8.4	83.9▲3.1
`Chain`	83.0▲2.6	88.3▲6.0	74.1▲3.0	93.3▲0.2	87.2▲0.6	81.4▲10.0	84.5▲3.7
`Tree`	81.0▲0.6	85.2▲2.9	71.2▲0.1	93.7▲0.6	88.9▲2.3	80.5▲9.1	83.4▲2.6
`Complete`	82.4▲2.0	80.1▼2.2	73.0▲1.9	94.5▲1.4	84.0▼2.6	79.0▲7.6	82.2▲1.4
`Random`	84.3▲3.9	86.9▲4.6	76.5▲5.4	94.1▲1.0	87.5▲0.9	82.7▲11.3	85.3▲4.5
`LLM-Debate`	85.0▲4.6	91.4▲9.1	77.7▲6.6	96.4▲3.3	90.1▲3.5	84.7▲13.3	87.5▲6.7
`AgentPrune`	85.1▲4.7	91.1▲8.8	80.5▲9.4	94.7▲1.6	90.6▲4.0	86.8▲15.4	88.1▲7.3
`AgentDropout`	85.6▲5.2	91.7▲9.4	80.9▲9.8	95.6▲2.5	91.0▲4.4	86.0▲14.6	88.5▲7.7
`G-Designer`	86.9▲6.5	93.8▲11.5	81.6▲10.5	96.5▲3.4	93.1▲6.5	88.3▲16.9	90.0▲9.2
`ARG-Designer`	89.5▲9.1	94.4▲12.1	86.5▲15.4	98.9▲5.8	95.6▲9.0	91.7▲20.3	92.8▲11.9
FRAME	**91.9**▲11.5	**95.9**▲13.6	**86.9**▲15.8	**99.6**▲6.5	**96.2**▲9.6	**92.6**▲21.2	**93.8**▲13.0
Open-source backbone: `Qwen3-8B`							
`Vanilla`	72.8	88.2	82.7	96.7	93.4	81.4	85.8
`CoT`	76.8▲4.0	91.8▲3.7	83.1▲0.4	97.7▲1.0	93.6▲0.2	87.6▲6.2	88.4▲2.6
`SC (CoT)`	77.3▲4.6	92.5▲4.3	84.1▲1.5	97.7▲1.0	93.8▲0.4	87.1▲5.7	88.8▲2.9
`AgentPrune`	78.4▲5.7	92.8▲4.6	85.0▲2.4	98.3▲1.7	94.9▲1.5	87.6▲6.2	89.5▲3.7
`G-Designer`	78.7▲5.9	93.1▲4.9	84.5▲1.8	98.1▲1.5	95.0▲1.6	88.4▲7.0	89.6▲3.8
`ARG-Designer`	76.7▲3.9	89.2▲1.0	84.0▲1.3	97.8▲1.1	93.6▲0.2	86.8▲5.4	88.0▲2.2
FRAME	**84.9**▲12.1	**93.7**▲5.5	**86.9**▲4.2	**99.2**▲2.5	**95.6**▲2.2	**91.7**▲10.3	**92.0**▲6.1

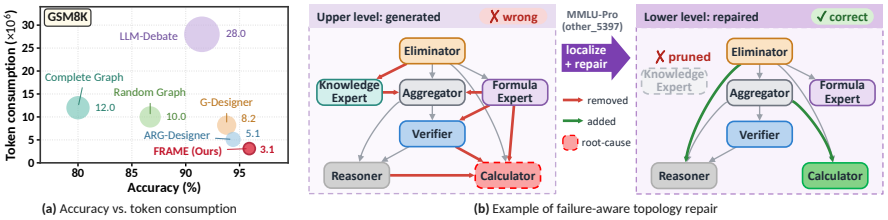
in the backbone LLM, each sampled topology is independently executed $M = 3$ times, and the resulting task rewards are aggregated for policy optimization. During inference, the learned node- and edge-policy parameters are fixed, and stochastic edge sampling is disabled to obtain a deterministic topology (more details on training and inference pipelines are given in Appendix C.2). Additional implementation details such as topology sparsity, the construction and size of the feasible role-repair set, and the hyperparameters, are provided in Appendix D.3 and D.5.

Discussion on Results. Table 1 reports results with GPT-4o (`gpt-4o-2024-08-06`) and `Qwen3-8B` (Yang et al., 2025) as LLM backbones, while Table 2 reports results with `Qwen3-8B` alone for complex datasets. Across all eight benchmarks, FRAME consistently achieves the strongest performance, outperforming the best-performing baselines. The gains are particularly pronounced on challenging benchmarks: on LCB and MMLU-Pro, FRAME improves over `ARG-Designer` by 17.0% and 18.2%. These improvements assert importance of conditioning topology evolution on execution states. Notably, the largest gains arise on tasks where reasoning failures are more likely to accumulate across agents, highlighting the potential importance of failure-aware topology repair.

Token and Parameter Efficiency. We evaluate the efficiency of FRAME along two dimensions: *token consumption* and *parameter cost*. Figure 3(a) compares accuracy against total token usage on GSM8K. FRAME achieves the strongest accuracy–efficiency trade-off, attaining 95.9% accuracy with only 3.1M tokens. Compared with `ARG-Designer` (94.4%, 5.1M tokens), this represents a 39.2% reduction in token consumption while also improving accuracy. Importantly, the reported token count for FRAME reflects the complete inference pipeline, including agent executions, failure-

Table 2: Results on LCB (Pass@1) and MMLU-Pro (Accuracy) with `Qwen3-8B` backbone.

Dataset	Vanilla	CoT	SC (CoT)	AgentPrune	G-Designer	ARG-Designer	FRAME
LiveCodeBench	47.0	49.0▲2.0	53.0 ▲6.0	53.0▲6.0	54.0▲7.0	52.0▲5.0	**69.0**▲22.0
MMLU-Pro	59.8	59.9▲0.1	63.9 ▲4.1	56.6▼3.2	57.1▼2.7	57.7▼2.1	**75.9**▲16.1



(a) Accuracy vs. token consumption (b) Example of failure-aware topology repair

Figure 3: **(a).** Token efficiency on GSM8K. **(b).** Topology without and with intervention.

repair calls, and judge calls, enabling a fair end-to-end comparison. Further analysis of token efficiency and parameter overhead is provided in Appendix D.5.

5.1 Ablation Study

We perform targeted ablations to isolate the contribution of the three central design choices in FRAME: *failure-triggered intervention*, *RFF−UCB role selection*, and *execution-state conditioning*.
Q1. Is explicit failure-triggered intervention necessary? We first isolate the contribution of the lower-level repair mechanism by comparing FRAME with a variant that disables intervention after topology construction. As shown in Table 3, removing the lower level reduces performance from 75.9% to 73.9% on MMLU-Pro and from 69.0% to 66.0% on LiveCodeBench. This indicates topology construction alone is insufficient particularly for challenging datasets. Additionally, in Figure 3(b) we illustrate this behavior on an MMLU-Pro example, where a localized role intervention changes the subsequent topology and recovers the final prediction. More details in Appendix B.3.
Q2. Does nonlinear RFF-UCB improve role selection over linear UCB? To examine importance of non-linearity introduced in RFF−UCB, we conduct study with a linear variant LinUCB (**w/o RFF-UCB**). Observe from Table 3, RFF−UCB consistently yields stronger task performance, indicating that a linear utility model is insufficient to capture the interaction between the task, selected roles, and states. Additional analysis on the RFF dimension, resulting topology differences is provided in Appendix D.4 and D.5.

Table 3: Ablation study on FRAME.

VARIANT	MMLU-Pro	LiveCodeBench
FRAME	**75.9**	**69.0**
w/o Lower-Level	73.9	66.0
w/o RFF-UCB	74.6	66.0
w/o Execution states	72.5	63.0

Q3. Does execution-state conditioning improve both topology quality and efficiency? Finally, we remove execution-state information from the autoregressive construction process, such that subsequent roles and edges are selected using only the task and partial topology. Table 3 shows a consistent performance drop under this ablation, demonstrating that intermediate execution outcomes provide information beyond the original task specification. Figure 1 provides a qualitative example: state-conditioned construction adapts subsequent topology decisions to the reasoning already produced, whereas ARG−Designer commits to roles without observing their realized contributions. This feedback also enables FRAME to avoid unnecessary additions or terminate construction earlier when the current graph is already sufficient, yielding improved token efficiency.

6 Conclusion

We introduced FRAME, a failure-aware and execution-conditioned framework for autoregressive topology learning in MAS. To jointly model topology construction and recovery, FRAME introduces a conditioned bilevel optimization problem the upper level incrementally constructs the collaboration graph from task and execution states, while the lower level performs targeted role repair when performance degrades below threshold. For efficient online construction, FRAME combines RFF-UCB-based role selection with a lightweight neural edge policy, and applies minimum-semantic intervention for topology correction. Extensive experiments across diverse benchmarks show that FRAME consistently improves task performance while reducing token usage, highlighting the value of treating topology generation and failure recovery as a unified closed-loop process.

AI USE STATEMENT

We used generative AI tools to assist with grammatical refinement, rephrasing, and language polishing of author-drafted text. All generated content was reviewed, revised, and validated by the authors before use. All experiments and reported results were produced using author-written code. The authors take full responsibility for the final content of this work, including its text, figures, claims, experimental methodology, and results.

ETHICS STATEMENT

This work uses only publicly available datasets and models and does not involve human subjects, personal data, or sensitive attributes. We adhered to applicable data and model licensing terms and to the ICLR Code of Ethics. We recognize potential risks associated with autonomous multi-agent LLM systems, including bias and misuse, and encourage responsible deployment with appropriate safeguards and human oversight.

REPRODUCIBILITY STATEMENT

We have taken several steps to facilitate the reproducibility of our work. The source code is provided as supplementary material accompanying the paper, and all datasets used in our experiments are publicly available. Experimental settings, model configurations, and implementation details are described in Section 5. Additional hyperparameters, prompts, and other implementation details required to reproduce the results are provided in Appendix D and Appendix E.

REFERENCES

Yasin Abbasi-Yadkori, Dávid Pál, and Csaba Szepesvári. Improved algorithms for linear stochastic bandits. *Advances in neural information processing systems*, 24, 2011.

Yixing Chen, Yiding Wang, Siqi Zhu, Haofei Yu, Tao Feng, Muhan Zhang, Mostofa Patwary, and Jiaxuan You. Multi-agent evolve: Llm self-improve through co-evolution. *arXiv preprint arXiv:2510.23595*, 2025.

Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of operations research*, 153(1):235–256, 2007.

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

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

Yu Ge, Linna Xie, Zhong Li, Yu Pei, and Tian Zhang. Spectrum-based failure attribution for multi-agent systems. *Proceedings of the ACM on Software Engineering*, 3(FSE):1888–1909, 2026.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, 2022.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Zhi Hong, Qian Zhang, Jiahang Sun, Zhiwei Shang, Mingze Kong, Xiangyi Wang, Yao Shu, and Zhongxiang Dai. Maspob: Bandit-based prompt optimization for multi-agent systems with graph neural networks. *arXiv preprint arXiv:2603.02630*, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

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

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)*, pp. 14165–14178, 2023.

Eric Hanchen Jiang, Levina Li, Frank Wan, Xiao Liang, Sophia Yin, Yuchen Wu, Xinfeng Li, Yizhou Sun, Wei Wang, Kai-Wei Chang, et al. Dynamic generation of multi llm agents communication topologies with graph diffusion models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 38042–38060, 2026.

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

Shiyuan Li, Yixin Liu, Yu Zheng, Mei Li, Quoc Viet Hung Nguyen, and Shirui Pan. Ofa-mas: One-for-all multi-agent system topology design based on mixture-of-experts graph generative models. In *Proceedings of the ACM Web Conference 2026*, pp. 1333–1344, 2026b.

Zhenwen Liang, Ye Liu, Tong Niu, Xiangliang Zhang, Yingbo Zhou, and Semih Yavuz. Improving llm reasoning through scaling inference computation with collaborative verification. *arXiv preprint arXiv:2410.05318*, 2024.

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

Ming Ma, Jue Zhang, Fangkai Yang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Dover: Intervention-driven auto debugging for llm multi-agent systems. In *International Conference on Learning Representations*, volume 2026, pp. 28059–28087, 2026.

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 *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025.

Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20, 2007.

Ankur Sinha, Pekka Malo, and Kalyanmoy Deb. A review on bilevel optimization: From classical to evolutionary approaches and applications. *IEEE transactions on evolutionary computation*, 22 (2):276–295, 2017.

Siyu Wang, Ruotian Lu, Zhihao Yang, Yuchao Wang, Yanzhou Zhang, Lei Xu, Qimin Xu, Guojun Yin, Cailian Chen, and Xinping Guan. Agentconductor: Topology evolution for multi-agent competition-level code generation. *arXiv preprint arXiv:2602.17100*, 2026a.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems*, 33:5776–5788, 2020.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

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

Yutong Wang, Siyuan Xiong, Xuebo Liu, Wenkang Zhou, Liang Ding, Miao Zhang, and Min Zhang. Agentdropoutv2: Optimizing information flow in multi-agent systems via test-time rectify-or-reject pruning. *arXiv preprint arXiv:2602.23258*, 2026b.

Zhexuan Wang, Yutong Wang, Xuebo Liu, Liang Ding, Miao Zhang, Jie Liu, and Min Zhang. Agentdropout: Dynamic agent elimination for token-efficient and high-performance llm-based multi-agent collaboration. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 24013–24035, 2025.

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.

Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

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

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. In *International Conference on Learning Representations*, volume 2025, pp. 75389–75428, 2025a.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. Agentracer: Who is inducing failure in the llm agentic systems? In *International Conference on Learning Representations*, volume 2026, pp. 11377–11399, 2026.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025b.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, et al. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems. *arXiv preprint arXiv:2505.00212*, 2025c.

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

APPENDIX OVERVIEW

A Notation Summary **13**

B Additional Results **13**

B.1 Effect of LLM Judge Selection at Inference 13

B.2 Performance Across LLM Backbones 14

B.3 Failure-Repair Diagnostics 15

C Supplementary Technical Details **16**

C.1 Details on Failure-Aware Repair Learning 16

C.2 Algorithms 17

C.3 Candidate Role Generation 17

C.4 Ablation on State Awareness for Token Reduction 17

D Experimental Setup **18**

D.1 Dataset Statistics 18

D.2 Baselines 19

D.3 Implementation Details and Hyperparameters 20

D.4 Effect of the RFF Dimension 20

D.5 Topology Statistics and Early Termination 21

E Prompts **21**

E.1 Failure-Aware Role Perturbation 22

E.2 Corrector Prompts 23

E.3 Agent Role Descriptions 23

E.4 Judge Prompts for Inference-Time Stopping 25

A NOTATION SUMMARY

Table 4 summarizes the notation. Throughout, $\cos(\cdot,\cdot)$ denotes cosine similarity, $g(\cdot)$ denotes the frozen text encoder, $\|$ denotes concatenation, and $\mathbb{I}[\cdot]$ denotes the indicator function. Arguments of set-valued and reward-valued maps are written with commas.

B ADDITIONAL RESULTS

B.1 EFFECT OF LLM JUDGE SELECTION AT INFERENCE

Table 5 examines the sensitivity of FRAME to the choice of LLM judge at inference, while fixing the backbone model to `Qwen3-8B` and keeping all other components unchanged. Replacing the `Qwen3.6-35B-A3B` judge with smaller models such as `Qwen3.5-27B` and `Qwen3-8B` results in only moderate performance degradation across HumanEval, MMLU, and LiveCodeBench. Despite the reduced judge capacity, FRAME retains strong performance, indicating that its gains are not solely driven by the choice of inference-time judge. More importantly, under a controlled setting with comparable backbone LLMs and judge models, our method delivers substantial gains over its closest prior work, `ARG-Designer`, particularly on challenging benchmarks such as LCB and MMLU-Pro.

Prior work has employed LLM-based evaluators for inference-time verification and for providing reward signals over generated responses (Liang et al., 2024; Chen et al., 2025). However, the role of

Table 4: Notation used in FRAME.

Symbol	Meaning	Symbol	Meaning
q	task instance	$\mathcal{A}$	agent pool, $\lvert\mathcal{A}\rvert = N$
$a_i = (r_i, s_i)$	agent role and execution state	$\mathcal{R}, \mathcal{S}$	role and state configurations
$G = (\mathcal{V}, \mathcal{E})$	collaboration DAG	$\mathcal{G}(q, \mathcal{R}, \mathcal{S})$	admissible topology set
$G_t, \mathcal{S}_t$	accepted prefix and state at step t	o_t^{acc}	accepted answer of prefix G_t
$\mathcal{A}_t$	available agents, $\mathcal{A} \setminus \mathcal{V}_{t-1}$	$g(\cdot)$	frozen text encoder
$\mathbf{z}_q, \mathbf{z}_i$	task and role embeddings	$c_{i,t}, d_{i,t}$	relevance and diversity features
$\mathbf{x}_{i,t}$	context vector in $\mathbb{R}^2$	η	task versus state relevance weight
$\mathbf{\Phi}(\cdot), D$	random Fourier features and their number	$\mathbf{\Psi}, \mathbf{y}$	ridge statistics in $\mathbb{R}^{D\times D}$, $\mathbb{R}^D$
α	UCB exploration scale	$\widehat{\mu}_{i,t}, \widehat{\sigma}_{i,t}$	predicted utility and uncertainty
γ_t	shaped node-addition reward	$\mathbf{E}_{\mathrm{in},t}$	incoming-edge mask of the new node
p_{j,i_t}, θ_e	edge probability and edge-policy parameters	K_G, K_E, M	construction runs, edge masks, rollouts
λ_E	edge-count penalty	β_t	REINFORCE baseline
$\bar{J}(\cdot)$	mean task reward over M rollouts	τ	target performance threshold
$z(G)$	underperformance indicator	$C(a)$	semantic divergence screening score
F	number of screened suspects	$\mathcal{M}_{\mathrm{ver}}$	verification model
$\widehat{o}_a$	corrected output of agent a	J_a^{cf}	counterfactual reward after correcting a
$a^\star$	verified failure source	$\Omega_{\mathcal{R}}(G)$	feasible role-repair set
$\Delta_{\mathrm{sem}}, \epsilon_{\mathrm{sem}}$	semantic repair cost and its threshold	δ_E	inference edge threshold
$\mathcal{R}^\star, G^\star$	repaired roles and reconstructed topology	$\mathcal{B}$	validated repair (skill) bank
$\xi, T(\xi)$	construction trace and its length	$\ell_{\theta_e}(\xi)$	log-likelihood of edges in ξ
K_0	failed traces per repair episode	ν, κ	penalty strength, counterfactual retention
r_{repair}	effective repair reward	$U(\cdot)$	upper-level graph utility
$\mathcal{M}_{\mathrm{judge}}$	inference-time judge	$\widehat{J}_t$	binary judge score at step t

Table 5: Effect of the LLM judge used at inference, with `Qwen3-8B` as the backbone LLM. Metric is accuracy (%) for MMLU and MMLU-Pro, and Pass@1 for HumanEval and LCB.

LLM JUDGE	HumanEval	MMLU	LCB	MMLU-Pro
`Qwen3.6-35B-A3B`	91.7	84.9	69.0	75.9
`Qwen3.5-27B`	90.9	83.6	66.0	74.0
`Qwen3-8B`	90.9	81.7	65.0	71.2

the judge in FRAME is substantially more restricted: it neither participates in topology construction nor contributes additional reasoning or candidate selection. Instead, it serves only as a termination signal, determining whether the current execution has sufficiently solved the task, without access to ground-truth labels or answers. Consequently, the improvements of FRAME cannot be attributed to the capacity of the judge, but rather to its execution-conditioned topology construction and failure-aware repair mechanisms. The weak sensitivity to judge scale further indicates that FRAME can employ a lightweight judge without materially affecting performance.

B.2 PERFORMANCE ACROSS LLM BACKBONES

Table 6 evaluates the robustness of FRAME across different LLM backbones. FRAME consistently delivers strong performance across both open-source and proprietary models, demonstrating that its effectiveness is not specific to a particular backbone. Among the open-source models, `Qwen3-8B` and `Gemma 4` achieve comparable results, with `Gemma 4` performing slightly better on HumanEval. In contrast, `Llama3.1-8B` exhibits a more pronounced degradation, particularly

Table 6: Performance of FRAME across LLM backbones.

LLM	**HumanEval**	**MMLU**
`Qwen3-8B`	91.7	84.9
`Gemma 4`	92.5	84.9
`Llama3.1-8B`	76.0	81.1
`gpt-4o`	**92.6**	**90.5**

Table 7: Failure-repair diagnostics on MMLU-Pro. Training statistics summarize the construction of validated repairs, while inference statistics characterize their reuse on held-out test instances.

Statistic	**Value**
Training-time repair	
Repair attempts	31
Validated repairs	9
Failed repair attempts	22
Repair success rate	29.0%
Inference-time skill reuse	
Skill-bank entries	9
Skill retrieval events	24
Successful recoveries	9
Success rate given retrieval	37.5%

on code-generation tasks, suggesting that the benefits of topology optimization remain constrained by the underlying reasoning capability of the backbone. While `gpt-4o` provides the strongest overall performance, capable open-source backbones combined with FRAME can achieve comparable performance.

B.3 FAILURE-REPAIR DIAGNOSTICS

Table 7 provides a diagnostic analysis of the failure-aware repair mechanism on MMLU-Pro. During training, the lower level is activated for 31 failed episodes, of which 9 produce validated role corrections that are retained in the skill bank. This corresponds to a training-time repair success rate of 29.0%. At inference, the resulting skill-bank entries are retrieved 24 times across held-out test instances, with 9 retrieval events leading to successful recovery. This yields a 37.5% success rate conditioned on retrieval.

The results show that validated repairs learned during training can be reused across multiple unseen instances rather than being generated independently for each test query. Importantly, the skill bank is constructed exclusively from validated repairs obtained on the training split and remains fixed throughout inference. No test labels, reference answers, rewards, or newly generated repairs are used to modify its contents. During evaluation, FRAME only retrieves and applies previously validated role corrections based on semantic similarity. The ground-truth label is accessed only after the final topology has been fixed and is used solely to compute the reported evaluation metric. Thus, the inference-time repair mechanism reuses previously learned repair knowledge without adapting the skill bank to test-set feedback.

Figure 3(b) provides a qualitative example of the repair process on an MMLU-Pro instance (`other_5397`), making the underlying mechanism more concrete. The upper-level construction initially produces an unsuccessful topology. The lower level then localizes the *Calculator* as the intervention target through failure screening and counterfactual verification. After applying the validated role correction, the upper-level topology is reconstructed under the repaired role configuration. This reconstruction removes interactions associated with the failed execution, prunes the now-unnecessary *Knowledge Expert*, and introduces new communication paths that support the corrected reasoning flow. The resulting topology produces the correct prediction. This example illustrates that the lower level does not simply correct an individual agent output. Instead, the localized role intervention alters the execution context used by the upper level, allowing the subsequent team composition and communication structure to adapt to the identified failure.

C SUPPLEMENTARY TECHNICAL DETAILS

C.1 DETAILS ON FAILURE-AWARE REPAIR LEARNING

The edge-policy objective in equation 15 learns communication structures from the rewards obtained during the initial autoregressive topology construction. When $z(G) = 1$, the subsequent intervention and role repair provide an additional learning signal: they reveal whether the failure can be attributed to a specific agent and whether correcting its role leads to a successful topology. We therefore augment the edge-learning objective with a failure-aware repair loss.

Effective repair reward. Let $\mathcal{R}^\star$ denote the role configuration obtained by replacing $r_{a^\star}$ with the selected repair, and let $G^\star$ denote the topology reconstructed under $\mathcal{R}^\star$ by the upper level. Even when $a^\star$ is correctly identified, $G^\star$ may fail to reach τ. The output-level intervention $\widehat{o}_{a^\star}$ in equation 17 corrects the agent's output directly, whereas the repaired role must reproduce the corrected behavior through a new execution. To retain the evidence obtained from successful causal verification, we define the effective repair reward as

$$r_{\text{repair}} = \max\big(\bar{J}(q, G^\star),\, \kappa J^{\text{cf}}_{a^\star}\big)\,, \qquad \kappa \in [0, 1], \tag{19}$$

where $\bar{J}(q, G^\star)$ is the mean reward of $G^\star$ over M rollouts and $J^{\text{cf}}_{a^\star}$ is the counterfactual reward obtained by correcting the output of $a^\star$. The counterfactual reward therefore serves as a proxy for the strength of the causal verification, and κ controls how much of this signal is retained when the zero-shot role repair does not fully reproduce the counterfactual correction.

Repair loss. Let $\{\xi^{(b)}\}_{b=1}^{K_0}$, with $1 \le K_0 \le K_G$, denote the upper-level construction traces that fall below τ in the current repair episode, and let $\xi^\star$ denote the trace that produced $G^\star$. We define

$$\mathcal{L}_{\text{penalty}} = \frac{\nu J^{\text{cf}}_{a^\star}}{K_0} \sum_{b=1}^{K_0} \ell_{\theta_e}\Big(\xi^{(b)}\Big)\,, \qquad \nu > 0, \tag{20}$$

$$\mathcal{L}_{\text{repair}} = \mathcal{L}_{\text{penalty}} - r_{\text{repair}}\, \ell_{\theta_e}(\xi^\star)\,. \tag{21}$$

Because the objective is minimized and ℓ_{θ_e} is a log-likelihood, the first term lowers the probability of the edge decisions that led to the failure, weighted by $J^{\text{cf}}_{a^\star}$ so that failures with strongly verified causes are penalized more. The second term raises the probability of the edge decisions in $\xi^\star$ in proportion to r_{repair}, which combines the empirical performance of the reconstructed topology with the counterfactual evidence from verification. When several repairs reach τ, $\xi^\star$ is the trace of the one with minimum semantic cost. When none does, $\xi^\star$ is the trace of the highest-reward candidate, and the floor in equation 19 keeps its reinforcement weight at least $\kappa J^{\text{cf}}_{a^\star}$.

Complete objective. The repair loss is defined only when a failure source has been verified. The complete training objective is

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{edge}} + z(G)\, \mathbb{I}[a^\star \text{ verified}]\, \mathcal{L}_{\text{repair}}, \tag{22}$$

where $\mathcal{L}_{\text{edge}}$ in equation 15 provides the standard reward-driven learning signal during autoregressive topology construction, and $\mathcal{L}_{\text{repair}}$ is active only when the topology underperforms and a failure source is verified. The RFF-UCB agent-selection policy remains gradient-free and is updated separately through its sufficient statistics $(\mathbf{\Psi}, \mathbf{y})$.

The two losses therefore play complementary roles. $\mathcal{L}_{\text{edge}}$ learns which communication structures yield high utility during normal topology construction, while $\mathcal{L}_{\text{repair}}$ uses verified failures to suppress the communication decisions associated with the failed execution and to reinforce the structures that emerge after successful role correction. This allows failure information from the lower-level problem to directly shape upper-level topology learning.

Importantly, the gains of FRAME are not contingent on the repair mechanism alone. As shown by the **w/o Lower-Level** ablation in Table 3, FRAME continues to substantially outperform `ARG-Designer` on the challenging benchmarks reported in Table 1, demonstrating that the benefits of execution-conditioned topology construction persist even in the absence of lower-level repair.

Training versus inference. During training, incoming edges are sampled as $e_{j,i_t} \sim \text{Bernoulli}(p_{j,i_t})$ to enable reward-based optimization of the edge policy. At inference, the RFF-UCB

statistics $(\mathbf{\Psi}, \mathbf{y})$ and the edge-policy parameters θ_e are frozen, and edges are selected deterministically as $e_{j,i_t} = \mathbb{I}[p_{j,i_t} \geq \delta_E]$, with no Bernoulli sampling or reward-based candidate selection. Generating new role candidates requires the reference answer through the perturbation prompt (Figure 5), which is unavailable for unseen tasks, so no new repairs are generated at test time. Instead, when the judge does not accept the constructed topology, FRAME reuses only repairs that were validated during training and stored in the skill bank $\mathcal{B}$. Suspects are ranked and verified as in training, with the judge score $\widehat{J}$ of equation 23 replacing the reward $\bar{J}$ wherever the latter appears, and a retrieved repair is applied only if the reconstructed topology is accepted by the judge (Algorithm 2).

C.2 ALGORITHMS

Algorithm 1 presents the training procedure for a single task, and Algorithm 2 describes inference. During evaluation, the ground-truth label is never exposed to the topology construction process, since using it to guide intermediate decisions would introduce test-time label leakage. Because the reward $\bar{J}$ is therefore unavailable at inference, we use an LLM judge $\mathcal{M}_{\text{judge}}$ in its place. After each construction step, the judge receives the task and the accepted output and replies with a single control token, `[STOP]` or `[CONTINUE]`, using the prompts in Figure 9. We map the reply to a binary score

$$\widehat{J}_t = \mathbb{I}[\mathcal{M}_{\text{judge}}(q, o_t^{\text{acc}}) = \texttt{[STOP]}] \in \{0, 1\}, \tag{23}$$

and construction stops once $\widehat{J}_t = 1$ or no roles remain. A reply that does not begin with a recognized control token is treated as `[CONTINUE]`. The judge score is used only to decide whether further topology expansion or inference-time repair is necessary. The ground-truth label is accessed only after the final topology $G^\star$ has been fixed, and is used solely to compute the reported evaluation metric J.

C.3 CANDIDATE ROLE GENERATION

Once the responsible agent $a^\star$ is identified, candidate corrections of its role are generated with a single LLM call using the perturbation prompt in Figure 5, with $\{\text{num_candidates}\}$ set to K. The prompt receives the task, the reference answer, the current role descriptions, the best answer obtained so far with its pass rate, and the intermediate agent outputs in execution order. It returns K variants $\widetilde{r}_{a^\star}$ of the responsible role, ordered from the smallest to the largest change, together with a skill category that serves as the retrieval key for the skill bank $\mathcal{B}$. Because the prompt requires the reference answer, candidate generation is performed only during training.

The prompt enforces two properties required by the lower-level problem. First, each variant is a self-contained role description rather than a instruction, so it can directly replace $r_{a^\star}$ in $\mathcal{R}$. Second, each variant preserves the core function of the original role, which keeps the semantic cost $\Delta_{\text{sem}}(\widetilde{r}_{a^\star}, r_{a^\star})$ small and prevents the repair from becoming a task-specific rewrite. Although the prompt may return variants for several agents, only those for $a^\star$ are retained. Candidates whose semantic cost exceeds a threshold ϵ_{sem} are discarded before evaluation, which avoids a full topology reconstruction for candidates that are unlikely to transfer across tasks.

C.4 ABLATION ON EXECUTION STATE AWARENESS FOR TOKEN REDUCTION

A key feature of FRAME is that each partial topology is executed before making the next structural decision, allowing subsequent decisions to condition on the current execution state. This enables FRAME to terminate topology construction as soon as the task is sufficiently solved, avoiding unnecessary agent invocations and downstream communication. As shown in Table 11, 89.0% of HumanEval tasks and 88.3% of GSM8K tasks terminate after a single agent, highlighting the resulting reduction in token consumption.

Execution state awareness also improves the quality of the learned topology rather than serving only as an efficiency mechanism. As shown in Table 3, removing execution-state feedback (**w/o Execution states**) reduces performance from 75.9% to 72.5% on MMLU-Pro and from 69.0% to 63.0% on LiveCodeBench. These results indicate that execution feedback serves a dual role in FRAME: it enables early termination when further collaboration is unnecessary and provides task-specific feedback for subsequent topology decisions when additional reasoning is required.

Algorithm 1 FRAME Training for Task q

Require: Task q, agent pool $\mathcal{A}$, role pool $\mathcal{R}$, threshold τ, skill bank $\mathcal{B}$

Upper Level: State-Aware Autoregressive Topology Learning

for $g = 1, \ldots, K_G$ **do**
  Initialize an empty topology prefix
  repeat
    Select the next role by RFF-UCB using Eqs. (5) and (8)
    Sample K_E edge masks using Eq. (12)
    Evaluate each candidate over M rollouts using Eq. (13)
    Extend the prefix with the highest-reward candidate
    Record the role reward and edge-policy signal using Eqs. (9) and (15)
  until reward reaches τ or no roles remain
end for
Select the best run as G
Compute the failure indicator z using Eq. (3)
If $z = 0$ then $G^\star \leftarrow G,\ \mathcal{L}_{\text{repair}} \leftarrow 0$

Lower Level: Failure-Aware Minimum-Intervention Topology Repair

if $z = 1$ **then**
  Rank the top-F suspects using Eq. (16)
  Verify suspects by counterfactual replay using Eq. (17)
  if a root cause $a^\star$ is verified **then**
    Retrieve a validated repair from $\mathcal{B}$ or generate K role corrections
    Reconstruct a topology for each correction using the upper level
    Retain candidates whose reward reaches τ
    Select the feasible repair with minimum semantic cost using Eq. (18)
    Update $G^\star$ and cache the validated repair in $\mathcal{B}$
    Compute $\mathcal{L}_{\text{repair}}$ using Eqs. (20) and (21)
  end if
end if

Learning Updates

Update the edge policy using Eq. (22)
Update the RFF statistics using Eq. (10)
return $G^\star$

D EXPERIMENTAL SETUP

D.1 DATASET STATISTICS

Table 8 summarizes the benchmarks used in our evaluation. We follow the evaluation protocols of Li et al. (2026a) and Jiang et al. (2026) for MMLU, GSM8K, MultiArith, SVAMP, AQuA, and HumanEval. For MMLU-Pro and LiveCodeBench, we define fixed train and test splits that are used consistently across FRAME and all baselines, with the exact splits released in our code. Policy parameters are learned only from the designated training split, and all reported results are computed on held-out test instances that are never used for policy learning or skill-bank construction.

Table 8: Summary statistics of the evaluation datasets.

Category	Dataset	Answer type	Metric	#Train	#Test	License
General reasoning	MMLU	Multi-choice	Acc.	40	153	MIT
	MMLU-Pro	Multi-choice	Acc.	196	1,204	MIT
Math reasoning	GSM8K	Number	Acc.	40	1,319	MIT
	MultiArith	Number	Acc.	40	600	Unspecified
	SVAMP	Number	Acc.	40	1,000	MIT
	AQuA	Multi-choice	Acc.	40	254	Apache-2.0
Code generation	HumanEval	Code	Pass@1	40	164	MIT
	LiveCodeBench	Code	Pass@1	102	100	CC

Algorithm 2 FRAME Inference for Task q

Require: Task q, agent pool $\mathcal{A}$, role pool $\mathcal{R}$, frozen statistics $(\mathbf{\Psi}, \mathbf{y})$, frozen edge policy θ_e, edge threshold δ_E, skill bank $\mathcal{B}$, judge $\mathcal{M}_{\text{judge}}$, suspect count F

Upper Level: State-Aware Topology Construction

Initialize an empty topology prefix G
repeat
  Select the next role using RFF-UCB via Eq. (8)
  Select incoming edges as $e_{j,i_t} = \mathbb{I}[p_{j,i_t} \geq \delta_E]$
  Extend G and execute the updated topology to obtain o_t^{acc}
  Compute $\widehat{J}_t$ using Eq. (23)
until $\widehat{J}_t = 1$ or no roles remain
$G^\star \leftarrow G$

Lower Level: Repair by Skill-Bank Retrieval Only

if $\widehat{J}_t = 0$ **then**
  Rank the top-F suspects via Eq. (16)
  Verify the most likely source $a^\star$ by intervention via Eq. (17), with $\widehat{J}$ in place of $\bar{J}$
  if $a^\star$ is verified **then**
    Retrieve task-relevant validated repairs for $a^\star$ from $\mathcal{B}$
    for each retrieved repair **do**
      Apply the repair and reconstruct the topology
      Execute the repaired topology and score it with Eq. (23)
    end for
    Set $G^\star$ to the minimum-cost repaired topology with $\widehat{J} = 1$, if one exists
  end if
end if
Evaluate $G^\star$ against the ground-truth label to obtain J
return $G^\star$, J

D.2 BASELINES

We compare against three families of methods, all instantiated on the same backbone and agent pool as FRAME.

Single-agent prompting. `Vanilla` A single backbone LLM call with the raw task prompt. `CoT` (Wei et al., 2022) elicits step-by-step reasoning. `SC (CoT)` (Wang et al., 2022) samples multiple reasoning chains and aggregates them by majority vote.

Fixed multi-agent topologies. This family fixes the communication structure a priori and reuses it for every query. `Chain` arranges agents in a sequential pipeline, `Tree` in a hierarchical branching structure, `Complete` as a fully connected graph in which every agent sees every other, and `Random` samples a sparse directed graph over the pool (Qian et al., 2025). `LLM-Debate` (Du et al., 2023) runs multiple rounds of debate over a fully connected graph before aggregating.

Learned topology design. This family adapts the interaction graph rather than fixing it. `AgentPrune` (Zhang et al., 2025a) prunes redundant communication edges from a dense graph via a learned mask. `AgentDropout` (Wang et al., 2025) additionally removes low-contribution agents and edges during optimization. `G-Designer` (Zhang et al., 2024) generates a task-conditioned adjacency matrix with a graph autoencoder. `ARG-Designer` (Li et al., 2026a) generates the team and its edges autoregressively conditioned on the task, and is the closest prior work to FRAME.

Protocol. All baselines share the backbone, decoding configuration, agent pool, and role descriptions used by FRAME, with published hyperparameters where available. On `GPT-4o` we evaluate the full set of baselines. On `Qwen3-8B` and on the two harder benchmarks (LiveCodeBench, MMLU-Pro) we retain the single-agent prompting family and the three strongest topology designers (`AgentPrune`, `G-Designer`, `ARG-Designer`), since the fixed topologies are dominated by the learned ones in every block where both are measured. The complete implementation, including baseline configurations and experimental code, is provided in the supplementary material.

D.3 Implementation Details and Hyperparameters

Table 9 summarizes the hyperparameters shared across all benchmarks, organized by the corresponding component of FRAME. Table 10 reports the benchmark-specific settings, namely the agent pool size N and the text encoder.

Reward definition. The task reward J is binary correctness for all benchmarks, so $\bar{J}$ is the empirical pass rate over M rollouts and lies in $\{0, 1/M, \ldots, 1\}$. This granularity is the reason M interacts with τ. With $M = 3$ and $\tau = 0.5$, a topology is treated as successful when at least two of its three rollouts are correct, and any threshold in $(1/3, 2/3]$ would induce the same behavior.

Table 9: Hyperparameters of FRAME shared across benchmarks. Benchmark-specific settings are listed in Table 10.

Component	Symbol	Value	Description
Representation			
Encoder width	d	384 / 768	MiniLM for text, UniXcoder for code
RFF dimension	D	{2, 4, ..., 32}	random Fourier feature count
Construction			
Construction runs	K_G	3	per training task
Edge masks per step	K_E	3	candidate incoming-edge configurations
Rollouts per candidate	M	3	repeated executions per candidate graph
Repair			
Performance threshold	τ	0.5	underperformance trigger
Screened suspects	F	2	top-F by $C(a)$
Generated candidates	K	4	role variants per episode
Counterfactual retention	κ	0.3	floor coefficient in equation 19
Verification model	$\mathcal{M}_{\text{ver}}$	`Qwen3.6-35B-A3B`	used in equation 17
Inference			
Judge model	$\mathcal{M}_{\text{judge}}$	`Qwen3.6-35B-A3B`	used in equation 23

Table 10: Per-benchmark settings.

Benchmark	Pool size N	Encoder
MMLU	6	MiniLM
MMLU-Pro	7	MiniLM
GSM8K	4	MiniLM
MultiArith	4	MiniLM
SVAMP	4	MiniLM
AQuA	4	MiniLM
HumanEval	5	UniXcoder
LiveCodeBench	5	UniXcoder

D.4 Effect of the RFF Dimension

Figure 4 examines the sensitivity of FRAME to the RFF dimension D used in the RFF-UCB role-selection module. The results exhibit the expected bias-variance trade-off: very small D limits the fidelity of the kernel approximation, whereas increasing the feature dimension improves the quality of the uncertainty-aware selection policy, with performance peaking around $D = 12$. Beyond this point, the additional representational capacity does not translate into further gains and can reduce stability because the online estimator must fit a higher-dimensional feature space from a limited number of observations. Importantly, performance remains relatively stable across a broad range of moderate D, indicating that FRAME is not critically dependent on precise tuning of the RFF dimension. This behavior is desirable for an online topology-learning setting, where a compact feature representation should provide sufficient expressiveness while preserving reliable uncertainty estimation and low computational overhead.

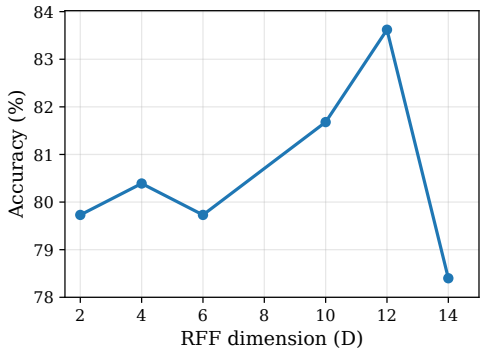


Figure 4: Effect of RFF dimension (D) on performance for MMLU dataset.

D.5 Topology Statistics and Efficiency Analysis

Table 11 reports realized topology statistics at inference with the `Qwen3-8B` backbone. These numbers are not directly comparable to the efficiency results in the main paper, which are measured with `GPT-4o`. Under `Qwen3-8B`, FRAME produces substantially sparser topologies than `ARG-Designer`. On HumanEval, the mean team size decreases from 4.04 to 1.48 and the mean edge count from 3.42 to 0.55. On GSM8K, they decrease from 3.81 to 1.07 and from 3.16 to 0.05, respectively. This yields token reductions of 30.2% on HumanEval and 64.2% on GSM8K, while the FRAME itself is $4.6\times$ smaller (0.66M vs. 3.01M parameters). Notably, #Parameters denotes the number of trainable parameters in the topology module for both methods.

A major source of this efficiency is adaptive single-agent termination. FRAME terminates after one agent on 89.0% of HumanEval and 88.3% of GSM8K instances, whereas `ARG-Designer` does not permit such early stopping. Since `Qwen3-8B` already attains strong single-agent performance on these benchmarks, part of the efficiency gain comes from avoiding unnecessary collaboration. This is nevertheless the intended effect of execution-conditioned topology construction. Evidence that FRAME also improves topology selection when additional collaboration is required is provided by its gains on the more challenging LiveCodeBench and MMLU-Pro benchmarks in Table 2.

Table 11: Realized topology statistics at inference for `Qwen3-8B` backbone.

Benchmark	Method	Mean $\lvert\mathcal{V}\rvert$	Mean $\lvert\mathcal{E}\rvert$	Single agent (%)	#Tokens	#Parameters
HumanEval	`ARG-Designer`	4.04	3.42	0	281336	3014085
	FRAME	1.48	0.55	89.0	196339	657810
GSM8K	`ARG-Designer`	3.81	3.16	0	5769622	3014085
	FRAME	1.07	0.05	88.3	2065094	657810

E Prompts

This appendix lists every prompt used in FRAME, grouped by where it acts in the pipeline. Two prompts support the lower-level repair during training. The corrector prompts (Figure 6) produce the corrected agent output used for counterfactual verification, and the perturbation prompt (Figure 5) generates candidate role repairs for the verified failure source. The agent role descriptions (Figures 7 and 8) define the role pool from which the topology designer composes a multi-agent system. The judge prompts (Figure 9) are used only at inference, where they decide after each construction step whether the system has already produced a correct answer and can stop. In all prompts, fields shown in {colored braces} are filled at runtime.

E.1 Failure-Aware Role Perturbation

Once the lower-level repair has verified a failure source $a^\star$ on a training task, we query an LLM with the prompt in Figure 5 to propose localized corrections to its role. The prompt receives the task, the reference answer, the current role descriptions, the best answer produced so far together with its pass rate, and the intermediate outputs of every agent in execution order. The model first assigns the task a single skill category, then attributes the failure to specific agents by comparing their outputs against the reference, and finally returns {num_candidates} variants of each implicated agent's role. The rules constrain these variants to be minimal, semantically close to the original, and ordered from smallest to largest change, so that perturbations refine a role rather than replace it. If no individual agent is at fault, the prompt falls back to the agent responsible for final aggregation. As described in Appendix C.3, FRAME sets {num_candidates} to K and retains only the variants proposed for $a^\star$.

Failure-Aware Role Perturbation Prompt

You are optimizing a multi-agent system that failed to solve the following task.

TASK:
{task_text}

CORRECT ANSWER:
{correct_answer}

CURRENT AGENT ROLES AND DESCRIPTIONS:
{role_descriptions}

BEST ANSWER PRODUCED SO FAR (pass rate: {best_rate:.2f}):
{best_answer}

INTERMEDIATE AGENT OUTPUTS (in order of execution):
{intermediate_states}

Your job:

0. Assign ONE short snake_case category naming the core skill this task needs (e.g., dynamic_programming, graph_traversal, greedy, string_parsing, math, simulation, binary_search). Choose the single best fit.
1. Analyze each agent's output against the correct answer and the task.
2. Identify ONLY the agents whose reasoning was incorrect, incomplete, or actively led the system toward the wrong answer.
3. For each such agent, propose exactly {num_candidates} SMALL variations of their role description that would correct the identified failure.

Rules:

- Do NOT include agents whose outputs were correct or on-track.
- Keep changes MINIMAL: adjust focus, add task-specific constraints, or clarify ambiguity. Do NOT change the fundamental nature of a role.
- Each variant must be semantically close to the original description.
- Order variants from smallest to largest change.
- Variants must be self-contained role descriptions, not instructions or diffs.

Output format:

Return ONLY a valid JSON object. No markdown, no explanation, no extra text:

```
{
  "category": "<snake_case_category>",
  "roles": {
    "RoleName": [
      "variant 1 description",
      "variant 2 description",
      "variant 3 description",
      "variant 4 description"
    ]
  }
}
```

Only include roles that need to change. If all agents reasoned correctly but the system still failed, include the agent most responsible for the final answer aggregation.

Figure 5: Prompt used to generate failure-aware role perturbations. Colored fields in braces are filled at runtime.

E.2 Corrector Prompts

For failed trajectories, we employ task-specific corrector prompts to repair the generated response. Within the lower-level repair, the verification model $\mathcal{M}_{\text{ver}}$ applies them to a suspect agent's output to obtain the corrected output $\widehat{o}_a$ in equation 17, whose counterfactual reward J_a^{cf} determines whether a is a verified failure source. Datasets belonging to the same task category share the same corrector prompt, shown in Figure 6. Each prompt informs the model that the previous attempt was incorrect and fixes the output format, so that the corrected response can be parsed by the same evaluator as the original. The general reasoning prompt requires a single option letter (MMLU, MMLU-Pro). The mathematical reasoning prompt requires a final numeric answer and is applied to all four math benchmarks. Its wording refers to grade-school word problems because it was designed for GSM8K, but the same prompt is used unchanged for AQuA. The code generation prompt requires a single Python code block without explanation, so the output can be executed directly against the test cases (HumanEval, LiveCodeBench).

General Reasoning — MMLU, MMLU-Pro

You are an expert answering a multiple-choice question. The previous answer was incorrect. Reason step by step, then state the single correct option on its own line as: `The answer is (X).`

Mathematical Reasoning — GSM8K, MultiArith, SVAMP, AQuA

You are an expert mathematician. The previous solution to this grade-school math word problem was incorrect. Re-solve it carefully with concise step-by-step reasoning, then finish with the final numeric answer on its own line as: `The answer is <number>.`

Code Generation — HumanEval, LiveCodeBench

You are a code debugger. Fix the failed code to solve the given task correctly.

Output only valid Python code in a ```` ```python ```` code block. No explanations.

Figure 6: Task-specific corrector prompts used to obtain corrected agent outputs for counterfactual verification. Datasets within a task category share the same prompt.

E.3 Agent Role Descriptions

The role descriptions form the node vocabulary from which the topology designer builds each multi-agent system, and they are also the text that the perturbation prompt (Figure 5) edits.

Figure 7 presents the seven roles used for the multiple-choice MMLU-Pro benchmark, where agents collaborate to solve questions spanning 14 domains. Four roles are specialists matched to the type of difficulty a question poses: logical inference (Reasoning Agent), recall of an established fact (Knowledge Expert), numeric computation (Calculation Agent), and selection of the governing law or equation (Formula Expert). The remaining three act on the outputs of earlier agents: the Option Eliminator narrows the ten options, the Verifier checks the step the current answer depends on most, and the Aggregator resolves disagreements between agents. The Verifier is an ordinary agent in the topology and is distinct from both the verification model $\mathcal{M}_{\text{ver}}$ and the inference-time judge $\mathcal{M}_{\text{judge}}$. Every role ends with a single option letter, so any agent can serve as the final node of the topology.

Agent Roles — MMLU-Pro, 7 roles

Reasoning Agent. You take questions that are answered by working through the logic (inference, argument, elimination by principle) rather than by recalling a fact or running a calculation. Read the question and all options and reason from what is given to the answer one step at a time, making each step explicit. Many questions are simple enough that your reasoning settles them alone. You rely on other agents only

when the question needs specialized knowledge, a genuine calculation, or is close enough to warrant a second look. End with your answer as a single letter (A–J).

Knowledge Expert. You take questions whose answer depends on a specific piece of established knowledge (a law or statute, a historical event, a scientific mechanism or property, a clinical fact, a definition, a named theorem or result) in any subject. Identify the exact fact the question rests on, state it, and apply it to the options with a precision that general reasoning would miss. If the question needs no such fact and is better reasoned out, give your best judgment rather than forcing a knowledge angle. State your option as a single letter (A–J).

Calculation Agent. You take questions that come down to computing a value (arithmetic, algebra, probability, unit conversion, numeric evaluation) in any subject, including quantitative economics, business, and finance. Set up the computation from the quantities given, carry units and signs through each step, and produce the number, guarding against arithmetic slips, mismatched units, and using a difference where a ratio is required. If the question involves no real computation, give your best judgment rather than inventing arithmetic. Report the value you reach as a single letter (A–J).

Formula Expert. You take questions where the difficulty is choosing the correct law, theorem, or governing equation and applying it correctly, most common in physics, chemistry, and engineering. Name the principle that applies, state its correct form, confirm its preconditions hold for this question, and set up the relation before any numbers go in. Your role is selecting and correctly applying the right relation, distinct from the arithmetic that follows it. State the option that correct application points to as a single letter (A–J).

Option Eliminator. You are brought in when several of the ten options remain plausible and the difficulty is telling them apart. Working from the question and whatever the earlier agents have produced, rule out options one at a time, naming each one's specific defect: wrong magnitude, contradicts a known fact, a correct idea applied incorrectly, or a distractor baiting a common mistake. You need not name the winner. You return a smaller, cleaner set of survivors for the others to decide among. Report which options you eliminated and which remain, ending with the one you find strongest as a single letter (A–J).

Verifier. You are brought in to check reasoning or work that already exists, most valuable after a long derivation or a chain of argument where an error could hide. Re-derive or sanity-check the single step the current answer most depends on, and flag any factual, computational, or logical mistake, especially a correct idea that was executed wrongly. State whether the leading option survives your check, and if not, where the corrected work points, as a single letter (A–J).

Aggregator. You are brought in at the end, once other agents have answered and their conclusions differ, to choose between them. Weigh each answer by whether it rests on a correct fact, a correct calculation, or sound reasoning, and settle the disagreement on that basis rather than by counting votes. If the earlier agents already agree, confirm their answer. You are needed only when there is a genuine conflict to resolve. Give the final answer as a single letter (A–J).

Figure 7: Agent role descriptions used for the MMLU-Pro benchmark.

Figure 8 presents the five roles used for LiveCodeBench, designed to facilitate collaborative problem solving for its competitive programming tasks. The roles mirror a standard development loop: the Algorithm Designer produces a plan with complexity analysis, the Programming Expert implements it, the Test Analyst identifies edge cases the implementation may miss, the Bug Fixer corrects failing code, and the Project Manager synthesizes all outputs into a final solution. Every role that emits code is restricted to a single Python code block and must preserve the LeetCode `Solution` class or the STDIN/STDOUT interface, so that the output of any code-producing node can be executed directly.

Agent Roles — LiveCodeBench, 5 roles

Algorithm Designer. You are an expert algorithm designer for competitive programming. Your job is to read the problem statement carefully, identify the core algorithmic challenge (e.g., dynamic programming, graph traversal, binary search, greedy, two-pointer), choose the optimal algorithm and data structures, and produce a clear step-by-step algorithmic plan with time/space complexity analysis. You do NOT write final code. You output an algorithmic plan that the Programming Expert will implement.

Programming Expert. You are an expert competitive programmer. Given a problem statement and optionally an algorithmic plan from another agent, you write a complete, correct, and efficient Python solution. For LeetCode problems, implement the required method inside the `Solution` class. For STDIN problems (AtCoder, Codeforces), write a complete program reading from stdin and printing to

stdout. Pay attention to edge cases, integer overflow, and time limits. Use a Python code block to write your response, and do not include anything other than a Python code block in your response.

Test Analyst. You are a rigorous test analyst for competitive programming. Given a problem and proposed solution, identify edge cases the solution may miss: empty inputs, single elements, maximum constraints ($n = 10^5$, values up to 10^9), negative numbers, already-sorted inputs, all-duplicate inputs, overflow cases, and boundary conditions specific to the problem. Describe failures clearly and concisely so the Bug Fixer can correct the code. Do not rewrite the code yourself. Focus on identifying what is wrong.

Bug Fixer. You are an expert at debugging competitive programming solutions. Given a solution that fails some test cases, identify the exact bug (off-by-one error, wrong base case, incorrect comparison, integer overflow, wrong modular arithmetic, etc.), explain the fix clearly, and produce the corrected complete solution. For LeetCode problems, preserve the `Solution` class structure. For STDIN problems, preserve the stdin/stdout structure. Use a Python code block to write your response, and do not include anything other than a Python code block in your response.

Project Manager. You are a technical project manager coordinating a team solving a competitive programming problem. Review the problem statement and all previous agents' outputs. Synthesize their outputs into the best possible final solution. Your output must be complete working code. For LeetCode problems, output the complete `Solution` class. For STDIN problems, output a complete program reading from stdin. Use a Python code block to write your response, and do not include anything other than a Python code block in your response.

Figure 8: Agent role descriptions used for the LiveCodeBench benchmark.

E.4 Judge Prompts for Inference-Time Stopping

At inference, the judge $\mathcal{M}_{\text{judge}}$ is invoked after each topology-construction step to determine whether the current execution is sufficient or additional agents should be introduced (Algorithm 2). Given only the task and the candidate outputs generated so far, the judge returns a control decision, `[STOP]` or `[CONTINUE]`, optionally accompanied by a brief justification. This decision is mapped to the binary score $\widehat{J}_t$ in equation 23. Importantly, the judge has no access to ground-truth answers, reference solutions, or hidden test cases, its decision is based solely on the task specification and the current agent outputs.

We use dataset-specific prompts to reflect the structure of each task while keeping the stopping criterion deliberately conservative. The `mmlu` and `mmlu_pro` variants require a single, well-supported multiple-choice decision. The `gsm8k` and `multiarith` variants require a definite numeric answer supported by consistent reasoning, while the `humaneval` and `lcb` variants assess whether the generated code is complete, syntactically valid, and consistent with the required functionality or I/O specification. In all cases, the judge emits `[STOP]` only when the current solution is sufficiently well-formed and justified, and defaults to `[CONTINUE]` under uncertainty. The complete prompts are provided in Figure 9.

Judge Prompt — `AQuA, SVAMP, MultiArith`

You are a verifier for a multi-agent problem-solving system. You are given a TASK and the CANDIDATE ANSWERS produced by the agents so far. Decide whether the current answer is already correct and complete enough that no further agents are needed.

You do NOT have the reference solution. Judge only from the task and the candidate answers: is the reasoning valid, is the answer fully specified, and does it actually solve what was asked? When confident it is correct and complete, reply `[STOP]`. Otherwise reply `[CONTINUE]`. When in doubt, reply `[CONTINUE]`.

Reply with exactly one token, `[STOP]` or `[CONTINUE]`, optionally followed by a brief reason.

Judge Prompt — `LiveCodeBench`

You are a verifier for a multi-agent code-generation system. You are given a coding TASK (problem statement) and the CANDIDATE CODE produced by the agents so far. Decide whether the code is a complete, self-contained, syntactically valid solution that plausibly satisfies the problem's requirements.

You do NOT have the test cases or reference solution. Judge only from the problem statement and the code: does it define the required function or I/O, handle the described cases, and appear logically correct? If yes, reply `[STOP]`. If it is incomplete, buggy, a stub, or you are unsure, reply `[CONTINUE]`.

Reply with exactly one token, `[STOP]` or `[CONTINUE]`, optionally followed by a brief reason.

Judge Prompt — `Humaneval`

You are a verifier for a multi-agent code-generation system. You are given a coding TASK and the CANDIDATE CODE produced so far. Decide whether the code is a complete, valid solution that plausibly satisfies the described behavior.

You do NOT have the tests or reference solution. Judge only from the task and the candidate code. Reply `[STOP]` only if confident that the solution is correct and complete; otherwise reply `[CONTINUE]`. When in doubt, reply `[CONTINUE]`.

Reply with exactly one token, `[STOP]` or `[CONTINUE]`, optionally followed by a brief reason.

Judge Prompt — `GSM8K`

You are a verifier for a multi-agent math-solving system. You are given a math TASK and the agents' CANDIDATE SOLUTIONS so far. Decide whether a clear, correct final numeric answer has been reached with sound reasoning.

You do NOT have the correct answer. Judge only from the task and the candidate solutions: is there a definite final answer, and is it supported by valid reasoning? Reply `[STOP]` only if confident; otherwise reply `[CONTINUE]`. When in doubt, reply `[CONTINUE]`.

Reply with exactly one token, `[STOP]` or `[CONTINUE]`, optionally followed by a brief reason.

Judge Prompt — `MMLU`

You are a verifier for a multi-agent multiple-choice QA system. You are given a TASK consisting of a question with lettered options (A–D) and the agents' CANDIDATE ANSWERS so far. Decide whether the agents have converged on a single, clearly stated option that is well justified by correct reasoning.

You do NOT have the answer key. Judge only from the question and the candidate answers: have the agents committed to one specific option, do they agree, and does the justification for that option hold up? Reply `[STOP]` only when there is a clear, consistent, and well-reasoned choice of a single option. If the agents disagree, hedge between options, provide no definite option, or the reasoning is weak, reply `[CONTINUE]`. When in doubt, reply `[CONTINUE]`.

Reply with exactly one token, `[STOP]` or `[CONTINUE]`, optionally followed by a brief reason.

Judge Prompt — `MMLU Pro`

You are a verifier for a multi-agent multiple-choice QA system. You are given a TASK consisting of a challenging question with up to ten lettered options (A–J) and the agents' CANDIDATE ANSWERS so far. Decide whether the agents have converged on a single, clearly stated option that is well justified by correct reasoning.

You do NOT have the answer key. These questions may require multiple reasoning steps, so judge only from the question and the candidate answers: have the agents committed to one specific option, do they agree, and does the justification adequately work through the problem? Reply `[STOP]` only when there is a clear, consistent, and carefully reasoned choice of a single option. If the agents disagree, hedge between options, provide no definite option, or the reasoning omits necessary steps, reply `[CONTINUE]`. When in doubt, reply `[CONTINUE]`.

Reply with exactly one token, `[STOP]` or `[CONTINUE]`, optionally followed by a brief reason.

Figure 9: Judge prompts used by $\mathcal{M}_{\text{judge}}$ at inference. Box titles indicate the corresponding dataset-specific prompt key used in our implementation.