RUNTIME-EVOLVING COMMUNICATION TOPOLOGY DESIGNER FOR LLM-BASED MULTI-AGENT COLLABORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-based multi-agent systems (MAS) have demonstrated remarkable collective intelligence, largely enabled by carefully designed inter-agent communication topologies. However, existing methods typically determine communication topologies before execution, preventing them from adapting information flow to runtime messages and evolving reasoning states. This can result in both redundant communication and missed task-critical interactions. To address this limitation, we propose RE-Designer, a **R**untime-**E**volving Communication Topology **Designer** that formulates topology construction as a sequential decision-making process and optimizes it through reinforcement learning (RL). Specifically, RE-Designer incrementally constructs a directed acyclic graph by dynamically adding agents and communication edges based on the partially constructed graph and the runtime messages generated by preceding agents. Since optimizing these sequential decisions with sparse task-level rewards is challenging, we introduce an information-bottleneck-inspired process reward that promotes task-relevant communication while penalizing redundant information propagation. To further reduce the exploration space, we develop a three-stage training strategy that first learns structural and message-aware priors through supervised learning and then refines runtime topology decisions through online RL. Extensive experiments on multiple reasoning benchmarks demonstrate that RE-Designer consistently achieves superior task performance, with favorable accuracy–communication trade-offs.

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.

1 INTRODUCTION

Large language model (LLM)-based multi-agent systems (MAS) have emerged as a promising paradigm for solving complex reasoning tasks. By enabling multiple agents with specialized roles to coordinate through the exchange of intermediate reasoning outputs, such systems can leverage diverse perspectives and collectively derive more reliable solutions (Du et al., 2024; Qian et al., 2024; Chen et al., 2024b; Yang et al., 2026; Li et al., 2023; Liang et al., 2024). In this collaborative process, the communication topology governs how information flows among agents, how their contributions are integrated, and ultimately how effectively the system solves a task. Consequently, designing effective and efficient communication topologies has become a fundamental challenge in LLM-based MAS (Zhang et al., 2025b; Qian et al., 2025).

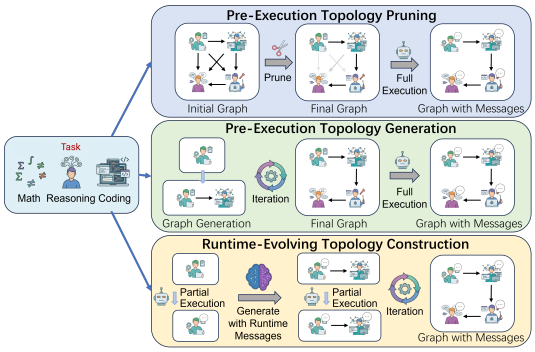


Figure 1: Comparison of pre-execution topology design and runtime-adaptive topology construction.

Early LLM-based MAS usually rely on manually designed communication topologies, such as chain, tree, star, fully connected, and random graphs (Qian et al., 2024; Hong et al., 2024; Wu et al., 2024; Qian et al., 2025). These fixed structures have limited capacity to adapt their information flow to the requirements of different tasks. Hence, recent studies have shifted toward automatically pruning or generating task-adaptive communication topologies. Representative methods explore graph learning (Zhang et al., 2025b), communication pruning (Zhang et al., 2025a; Wang et al., 2025b), autoregressive graph generation (Li et al., 2026a;b), and diffusion-based graph generation (Zhang et al., 2026; Jiang et al., 2026), substantially improving the flexibility and communication efficiency of LLM-based MAS.

Despite these advances, most existing topology design methods share a fundamental limitation: their communication topologies are determined prior to multi-agent execution, without considering the intermediate messages and reasoning states that emerge as agents solve the task. As a result, communication decisions cannot respond to the evolving reasoning process, potentially preserving unnecessary interactions while failing to establish new communication links when additional information becomes relevant. To overcome this limitation, as illustrated in Figure 1, we shift communication topology design from a pre-execution, fixed paradigm to a runtime-adaptive paradigm, where communication topologies are progressively constructed based on the intermediate information generated during reasoning.

Specifically, we propose RE-Designer, a **R**untime-**E**volving Communication Topology **Designer** for LLM-based multi-agent collaboration. RE-Designer reformulates topology construction as an in-execution autoregressive graph generation process, in which graph generation and agent reasoning are interleaved. Rather than fixing the entire topology before execution, RE-Designer progressively constructs the communication graph throughout the reasoning process, conditioning each subsequent edge decision on both the partial graph constructed so far and the runtime messages produced by previously executed agents.

This sequential formulation makes reinforcement learning (RL) a natural framework for optimizing the topology generation policy. However, learning such a policy from task outcomes alone is challenging due to sparse and delayed rewards: the outcome reward evaluates only the completed reasoning trajectory and provides little guidance on which agent should be invoked or which edge should be added under the current reasoning state. To provide denser supervision without introducing an additional latency-inducing LLM-based evaluator, we introduce a semantic weighting signal inspired by the information bottleneck principle (Tishby et al., 2000). The proposed signal rewards communication decisions that preserve task-relevant information while penalizing redundant information propagation, thereby facilitating the learning of an effective and communication-efficient runtime topology policy.

While the information-guided reward provides informative feedback for runtime communication decisions, exploring the combinatorial space of runtime topologies remains challenging. We therefore introduce a three-stage training strategy that progressively equips the topology policy with structural and message-aware priors before online policy optimization. In the first stage, we train the policy through supervised learning to imitate basic structural patterns of communication graph construction. In the second stage, we further train the policy on runtime messages collected from supervised execution trajectories, enabling it to encode intermediate reasoning outputs and make message-aware topology decisions. Finally, we refine the topology policy through online RL using both task rewards and the proposed information-guided reward, allowing it to explore more effective and communication-efficient runtime topologies.

Our contributions are summarized as follows:

- We formulate multi-agent communication topology construction as a runtime sequential decision process and develop RE-Designer, which interleaves topology construction with agent execution and sequentially selects agent roles, predecessor edges, and stopping decisions based on the evolving reasoning process.
- We introduce an information-bottleneck-inspired process reward that provides step-level guidance for topology optimization by promoting task-relevant information while discouraging redundant communication.
- Experiments across six reasoning benchmarks show that RE-Designer achieves the best average performance and a favorable accuracy–communication trade-off.

2 Related Work

Automatic Communication Topology Design Early LLM-based multi-agent systems rely on manually specified communication structures (Qian et al., 2024; Hong et al., 2024; Wu et al., 2024). Recent studies have explored automatic topology design to improve the flexibility and efficiency of multi-agent communication. AgentPrune and AgentDropout reduce communication redundancy through topology pruning, removing unnecessary edges or agents from collaboration structures (Zhang et al., 2025a; Wang et al., 2025b). G-Designer learns task-dependent communication graphs, while ARG-Designer formulates topology design as an autoregressive graph generation process that sequentially generates agents and connections (Zhang et al., 2025b; Li et al., 2026a). OFA-MAS further investigates unified topology generation across diverse tasks (Li et al., 2026b). More recently, RADAR and GTD introduce diffusion-based approaches for communication topology generation and optimization (Zhang et al., 2026; Jiang et al., 2026). However, most topology-design methods determine communication structures before execution. Among runtime-adaptive approaches, AnyMAC predicts the next agent and selects relevant context from previous outputs during execution (Wang et al., 2025a). RE-Designer instead explicitly maintains an evolving communication graph, jointly modeling partial topology and runtime messages for graph-construction decisions with process-level RL guidance.

Communication Value Modeling in Multi-Agent Systems Effective multi-agent collaboration depends not on the sheer volume of communication, but on whether the exchanged information is task-relevant and complementary. Recent studies on LLM-based MAS have begun to investigate how information diversity, disagreement, and interaction patterns affect collaborative reasoning (Chen et al., 2024a; Wu & Ito, 2025; Yang et al., 2026). Meanwhile, research on multi-agent RL under partial observability has long recognized the importance of assessing the value of agents' local information for effective communication and coordination (Wang et al., 2020a). Inspired by this general perspective, RE-Designer introduces an information-bottleneck-inspired reward to estimate the utility of runtime messages generated by individual LLM agents, thereby providing fine-grained guidance for edge decisions during runtime graph construction.

3 Problem Formulation

This section first defines the graph-based communication structure and execution process of LLM-based MAS, and then formulates runtime topology construction as a finite-horizon sequential decision process.

3.1 Communication Graph Structure

Given a task query q and a predefined role pool R, an LLM-based multi-agent system can be represented as a directed acyclic graph (DAG) $G = (V, E)$, where V is the set of agent nodes and E is the set of directed communication edges. Each node $v_i \in V$ represents one LLM agent execution with a role $r_i \in R$ and produces a response message m_i. Each directed edge $(v_j, v_i) \in E$ means that the message generated by v_j can be used when building the prompt for v_i, with the predecessor set of v_i defined as $\mathrm{Pa}(i) = \{v_j \mid (v_j, v_i) \in E\}$. The DAG constraint ensures that each agent only uses previously generated messages, giving a valid causal execution order.

3.2 Communication Pipeline

Each agent node receives a system prompt and a user prompt, denoted as $P_i = \{P^i_{\mathrm{sys}}, P^i_{\mathrm{user}}\}$, where P^i_{sys} contains the role instruction derived from r_i. The user prompt of v_i is built from the task query and the messages generated by its predecessor nodes: $P^i_{\mathrm{user}} = \{q, \{m_j \mid v_j \in \mathrm{Pa}(i)\}\}$. The corresponding LLM agent then produces its response message as $m_i = v_i(P_i) = v_i(P^i_{\mathrm{sys}}, P^i_{\mathrm{user}})$. Each edge in G therefore defines an explicit prompt-level dependency by deciding which previous messages are included in the user prompt of the current agent. Nodes are executed in the causal order of the graph, allowing later nodes to use real messages generated by earlier nodes.

After communication ends, an aggregation node receives the task query and all generated messages and produces a reference response $A_{\mathrm{ref}} \leftarrow \mathrm{Aggregate}(q, \{m_i \mid v_i \in V\})$. The aggregation function

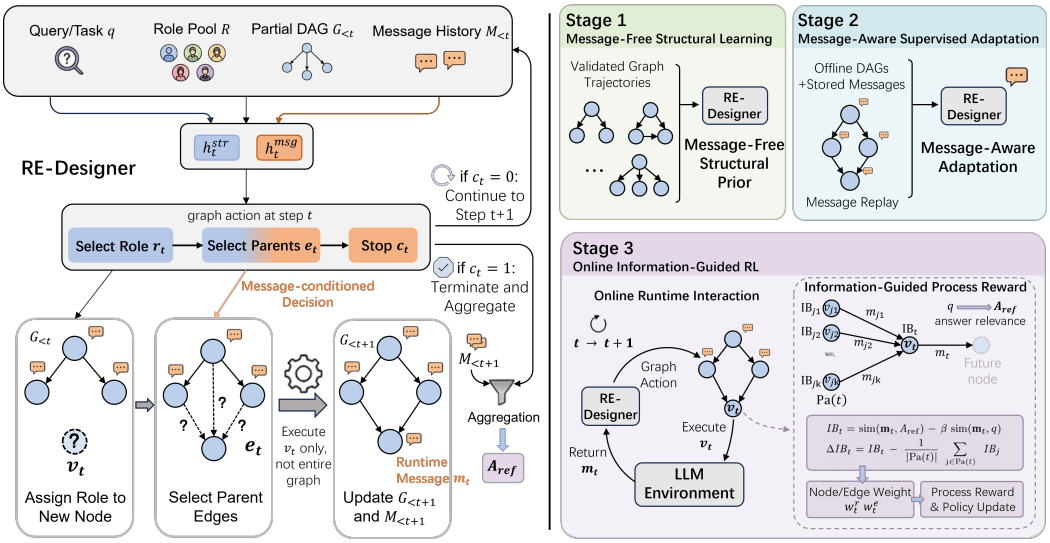


Figure 2: Overview of RE-Designer. Runtime topology construction is interleaved with agent execution, allowing messages to guide subsequent edge decisions. The framework uses three-stage training and an information-guided process reward.

can use majority voting, the output of the terminal agent, or the concatenation and consolidation of all agent messages. In this work, we use the last option to produce A_{ref}, from which the final answer a is extracted according to the task-specific evaluation protocol. The aggregation node is part of the standard MAS execution process. Its output is also used later to compute the information-guided process reward.

3.3 Runtime Topology Construction as a Sequential Decision Process

We formulate runtime topology construction as a finite-horizon Markov decision process $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}$ is the transition function defined by LLM-based MAS execution, $\mathcal{R}$ is the reward function, and γ is the discount factor.

At construction step t, the state is defined as $s_t = \{q, G_{<t}, M_{<t}\}$, where $G_{<t}$ is the partial graph built before step t, and $M_{<t} = \{m_1, \ldots, m_{t-1}\}$ is the set of available runtime messages. The action is defined as $a_t = (r_t, e_t, c_t)$, where $r_t \in R$ selects the role of the new node, $e_t \in \{0,1\}^{t-1}$ gives its predecessor mask, and $c_t \in \{0,1\}$ gives the stop decision.

At each step, the policy generates a new node and its incoming edges from the current state, calls the selected agent to obtain message m_t, and updates both the graph and the message history. When the policy stops or reaches the construction budget, the aggregation node generates A_{ref}, after which the final answer is extracted and the task reward is computed.

4 RE-Designer for MAS Collaboration

We introduce RE-Designer, which interleaves autoregressive graph construction with agent execution and conditions subsequent topology decisions on runtime messages. As illustrated in Figure 2, its three components are a runtime message-aware topology policy that adapts communication during execution, an information-guided process reward that promotes task-relevant communication while reducing redundancy, and a three-stage training process that learns structural and message-aware priors before online RL.

4.1 RUNTIME MESSAGE-AWARE TOPOLOGY POLICY

At construction step t, RE-Designer conditions on the query q, role pool R, partial graph $G_{<t}$, and message history $M_{<t}$. It determines the role r_t and incoming edges e_t of the next node, whose agent processes the selected predecessor messages to produce m_t. The recurrent states are then updated before the stop decision c_t determines whether construction continues.

The query, role descriptions, and runtime messages are embedded by the same frozen sentence encoder (Wang et al., 2020b). The query and role embeddings provide message-independent context, and message embeddings are incorporated into the message-aware path. Structural features $\psi(G)$ contain the node and edge counts, density, depth, average indegree, and remaining construction budget, providing topology context without requiring an additional graph encoder.

RE-Designer maintains two recurrent states across construction steps. The structural state h_t^{str} summarizes the query, selected roles, graph-structure history, and other message-independent context. The message-aware state h_t^{msg} separately summarizes generated messages and context from selected predecessors. Role selection uses the structural state, whereas edge selection additionally uses the message-aware state.

The topology policy is factorized into role, edge, and stop decisions:

$$\pi_\theta(a_t \mid s_t) = \pi_\theta^r(r_t \mid h_t^{\mathrm{str}}) \cdot \pi_\theta^e(e_t \mid r_t, h_t^{\mathrm{str}}, h_t^{\mathrm{msg}}) \cdot \pi_\theta^c(c_t \mid h_{t+1}^{\mathrm{str}}, h_{t+1}^{\mathrm{msg}}). \tag{1}$$

Here, π_θ^r selects the role of the next agent, π_θ^e generates its incoming edges, and π_θ^c determines whether another construction step is needed. The states h_{t+1}^{str} and h_{t+1}^{msg} are obtained after the current node and edges are added and the agent produces m_t. They are therefore available before the stop decision and serve as the recurrent states of the next step if construction continues. Following the Mixture-of-Experts (MoE) design used in recent topology generation models (Li et al., 2026b), RE-Designer uses lightweight MoE modules for role, edge, and stop prediction, with a GRU-based edge decoder operating recurrently over candidate predecessors.

The role policy determines which type of agent should be added next. The edge policy is the main message-aware routing component and predicts the predecessor mask from the selected role and both recurrent states. For each previous node $j < t$, the recurrent decoder estimates $p_{t,j}^e = \pi_\theta^e(e_{t,j} = 1 \mid h_t^{\mathrm{str}}, h_t^{\mathrm{msg}}, r_t, \mathbf{e}_{t,>j})$, where $\mathbf{e}_{t,>j}$ denotes the previously decoded edge decisions under the reverse predecessor order. When evaluating candidate predecessor v_j, the decoder incorporates its role representation, runtime message representation, and relative decoding position into its recurrent context. This recurrent decoding allows each edge decision to depend on both previously decoded edge decisions and the actual content produced by candidate predecessors.

After agent execution, the two states are updated through separate GRU-based paths. The structural state incorporates the selected role and post-update graph structure without runtime message embeddings. The message-aware state additionally incorporates the selected-predecessor context and newly generated message m_t. The updated states condition the current stop decision and subsequent topology decisions. Since m_t becomes available only after the current role and edges have been selected, it cannot affect the role or edges already selected at step t.

4.2 INFORMATION-GUIDED PROCESS REWARD

The runtime topology policy turns topology construction into a sequential decision process, where the effect of each node and edge decision may only become clear after later reasoning steps. The outcome reward provides trajectory-level feedback, but little guidance for intermediate topology decisions under the current reasoning state. RE-Designer therefore introduces an information-guided process reward that modulates the outcome reward across graph-construction steps according to the information contribution of each message.

Inspired by the information bottleneck principle (Tishby et al., 2000), we encourage runtime messages to keep task-relevant information while reducing redundant content. Since directly estimating mutual information over high-dimensional language representations is difficult (Poole et al., 2019), we use semantic similarity from a frozen sentence encoder as a proxy for task-relevant information (Reimers & Gurevych, 2019). For node t, we define the following information-bottleneck-

inspired score:

$$IB_t = \mathrm{sim}\left(e(m_t), e(A_{\mathrm{ref}})\right) - \beta\, \mathrm{sim}\left(e(m_t), e(q)\right), \tag{2}$$

where A_{ref} is the natural-language response generated by the aggregation node in the current rollout, q is the input query, $e(\cdot)$ is the same frozen sentence encoder used for runtime message representation, and $\mathrm{sim}(\cdot,\cdot)$ denotes semantic similarity. We require A_{ref} to include a brief rationale rather than only the final answer. This score is an information-bottleneck-inspired proxy rather than a direct optimization of the classical information bottleneck objective. The first term uses A_{ref} as a target-side proxy for task-relevant information, while the second term penalizes content that mainly repeats the input query. Note that A_{ref} becomes available only after the complete trajectory and is not the ground-truth answer. It is used only to compute the information score after the rollout and therefore does not affect decisions made within the same rollout.

To measure the new information introduced by node t, we compare its score with the average score of its parent messages:

$$\Delta IB_t = IB_t - \frac{1}{|\mathrm{Pa}(t)|} \sum_{j \in \mathrm{Pa}(t)} IB_j. \tag{3}$$

For a root node without predecessors, the query representation is used as the reference baseline. Since each node directly uses its selected predecessors, this parent-relative difference gives a local estimate of how much answer-relevant information is added by the current communication step. A positive ΔIB_t suggests that the current node adds answer-relevant information beyond the messages it receives.

The incremental information scores are normalized within each trajectory to obtain step-level information weights:

$$w_t = 1 + \eta \frac{\Delta IB_t}{\sum_{k=1}^{T} \max(0, \Delta IB_k) + \epsilon}. \tag{4}$$

Here, η controls the strength of information weighting, and ϵ ensures numerical stability. To ensure valid weighting, we clip the resulting values to the non-negative range so that $w_t \geq 0$. Positive information gains in the denominator stabilize the overall scale, while the numerator preserves the sign of ΔIB_t. Steps with larger contributions therefore receive higher weights, while less useful or redundant steps receive lower weights. Information weighting is only used for trajectories with correct final answers. For incorrect trajectories, information weighting is disabled and $w_t = 1$. Thus, the information signal only differentiates process rewards within successful trajectories.

For trajectory i, let $w_{i,t}$ denote the corresponding step-level weight. We apply it primarily to edge decisions and use separate weights for role, edge, and stop decisions; the exact decision-specific weighting is given in Appendix B.2.

Let R_i be the task reward of trajectory i, and let b_i be its variance-reduction baseline. We compute b_i using RLOO (Ahmadian et al., 2024), which provides a query-level baseline from other rollouts of the same query and accounts for differences in query difficulty. The policy loss is

$$\mathcal{L}_{\mathrm{policy}} = -\sum_{i,t} \sum_{a \in \{r,e,c\}} \left(w_{i,t}^{a} R_i - b_i\right) \log \pi_\theta^a\left(a_{i,t} \mid s_{i,t}\right). \tag{5}$$

The task reward evaluates the complete trajectory, while the information weights modulate the contribution of individual graph-construction steps within successful trajectories. The full information weight is applied to edge decisions, a reduced weight to role decisions, and no information weight to stop decisions.

4.3 Three-Stage Training

Directly training RE-Designer with RL is difficult because the policy must learn valid topology construction, message-aware topology adaptation, and task-level optimization at the same time. RE-Designer therefore uses three training stages to reduce the exploration space. Stage 1 learns basic topology generation without runtime message inputs. Stage 2 learns state updates and decision adaptation under real message conditions while keeping the structural prior. Stage 3 freezes the message-aware adaptation learned in Stage 2 and further optimizes the runtime topology policy.

The first two stages use offline supervised trajectories. We generate candidate communication graphs from different connectivity patterns, including chain, layered, mesh, and fully connected structures, and assign node roles from the candidate role pool of each task. Each candidate graph is tested through actual LLM-based multi-agent execution, and only trajectories with correct final answers are kept. The graph structure and real runtime messages generated during execution are stored together. When several correct trajectories are available for the same query, structurally compact trajectories are prioritized as the main supervision targets, while other correct trajectories are retained as additional examples to preserve topology diversity. We refer to this preference as the compact prior.

In **Stage 1**, RE-Designer uses message-free supervised behavior cloning on the offline trajectories. Runtime messages are not used in this stage. The policy learns role selection, predecessor-edge generation, and stopping from the task query and current structural history, which gives the model basic topology generation ability and transferable structural priors. The training objective is

$$\mathcal{L}_{\mathrm{SFT}} = \mathcal{L}_r + \lambda_e \mathcal{L}_e + \lambda_c \mathcal{L}_c + \lambda_{\mathrm{moe}} \mathcal{L}_{\mathrm{moe}}, \tag{6}$$

where $\mathcal{L}_r$, $\mathcal{L}_e$, and $\mathcal{L}_c$ are the role, edge, and stop prediction losses, and $\mathcal{L}_{\mathrm{moe}}$ is an auxiliary load-balancing loss for the prediction heads. Stage 1 is jointly trained across all datasets to learn transferable graph-structure priors.

In **Stage 2**, RE-Designer performs message-aware adaptation using the real execution messages stored in the offline trajectories. The trajectories are replayed in their original execution order, and the corresponding intermediate messages are added to the policy state so that the model learns how message history should affect subsequent topology decisions. To keep the structural prior learned in Stage 1, the message-independent policy components remain fixed, while only the message-conditioned pathway and its related adaptation parameters are updated. After this stage, RE-Designer preserves the structural prior learned in Stage 1 while adapting subsequent topology decisions to real runtime messages.

In **Stage 3**, RE-Designer performs online RL separately for each dataset. Unlike the first two supervised stages, the policy autonomously generates communication trajectories and obtains real runtime messages through actual agent execution. Since the message-conditioned pathway has already been adapted to runtime conditions in Stage 2, we keep it fixed during RL to separate message-aware state adaptation from topology optimization. The generated messages are still fully incorporated into the policy state, allowing subsequent topology decisions to remain message-aware. The task reward serves as the primary optimization signal, while the information-guided process reward differentiates the learning signals of topology decisions within successful trajectories. This design enables RL to focus on runtime topology optimization while preserving the structural prior and message-aware adaptation learned from supervised training.

5 Experiments

5.1 Experimental Setup

Datasets and Metrics. We evaluate RE-Designer on six benchmarks covering three task types: MMLU (Hendrycks et al., 2021) for general reasoning; GSM8K (Cobbe et al., 2021), MultiArith (Roy & Roth, 2015), SVAMP (Patel et al., 2021), and AQuA (Ling et al., 2017) for mathematical reasoning; and HumanEval (Chen et al., 2021) for code generation. We report accuracy on MMLU and the four mathematical reasoning datasets, and Pass@1 on HumanEval. The reported results are averaged over three runs.

Baselines. We compare RE-Designer with two groups of baselines. The first group contains single-agent methods, including Vanilla, Chain-of-Thought (CoT) (Wei et al., 2022), and self-consistent Chain-of-Thought (CoT-SC) (Wang et al., 2023). The second group includes automatic topology design methods, including DyLAN (Liu et al., 2024), GPTSwarm (Zhuge et al., 2024), AgentPrune (Zhang et al., 2025a), AgentDropout (Wang et al., 2025b), G-Designer (Zhang et al., 2025b), ARG-Designer (Li et al., 2026a), and OFA-MAS (Li et al., 2026b).

Implementation Details. RE-Designer and all reproduced baselines use Qwen3-8B (Yang et al., 2025) with thinking disabled as the agent backbone. All multi-agent methods perform a single graph

Table 1: Performance comparison on benchmarks. Changes are reported relative to Vanilla.

Method	MMLU	GSM8K	MultiArith	SVAMP	AQuA	HumanEval	Avg.
Vanilla	72.78	88.15	96.67	93.60	82.68	81.37	85.88
CoT	76.81↑4.03	91.81↑3.66	97.50↑0.83	93.40↓0.20	83.07↑0.39	87.58↑6.21	88.36↑2.49
CoT-SC	77.34↑4.56	92.48↑4.33	97.67↑1.00	93.83↑0.23	84.13↑1.45	87.08↑5.71	88.76↑2.88
DyLAN	78.43↑5.65	89.92↑1.77	97.67↑1.00	93.90↑0.30	86.22↑3.54	85.98↑4.61	88.69↑2.81
GPTSwarm	79.42↑6.64	93.27↑5.12	97.66↑0.99	94.75↑1.15	79.96↓2.72	88.12↑6.75	88.86↑2.98
AgentPrune	78.43↑5.65	92.78↑4.63	**98.33**↑1.66	94.90↑1.30	85.04↑2.36	87.58↑6.21	89.51↑3.63
AgentDropout	79.31↑6.53	93.21↑5.06	97.72↑1.05	94.85↑1.25	84.66↑1.98	88.06↑6.69	89.64↑3.76
G-Designer	78.65↑5.87	93.05↑4.90	98.13↑1.46	95.00↑1.40	84.52↑1.84	88.40↑7.03	89.63↑3.75
ARG-Designer	76.68↑3.90	89.18↑1.03	97.81↑1.14	93.57↓0.03	83.95↑1.27	86.78↑5.41	88.00↑2.12
OFA-MAS	79.91↑7.13	91.01↑2.86	98.21↑1.54	93.97↑0.37	84.11↑1.43	89.26↑7.89	89.41↑3.53
RE-Designer	**81.04**↑8.26	**93.48**↑5.33	98.10↑1.43	**95.17**↑1.57	**86.92**↑4.24	**92.28**↑10.91	**91.16**↑5.28

execution, with each selected agent invoked once. The maximum number of generated nodes is 6. Query, role, and message representations are encoded by a frozen `all-MiniLM-L6-v2` sentence encoder (Wang et al., 2020b) with an embedding size of 384. Stages 1 and 2 are jointly trained across all six datasets, while Stage 3 performs dataset-specific online RL using RLOO (Ahmadian et al., 2024). We use 40 training queries from each dataset throughout all three training stages. We set β to 2.0. Full dataset, supervised-trace, training, and inference details are provided in Appendix C.

5.2 Performance Comparison

Table 1 compares RE-Designer with single-agent methods and automatic topology design methods across six benchmarks. RE-Designer achieves the highest average score of 91.16%, outperforming the two best-performing baselines in our experiments, G-Designer and AgentDropout, by 1.53 and 1.52 percentage points, respectively. It ranks first on five benchmarks spanning general reasoning, mathematical reasoning, and code generation, showing that its gains are not concentrated on a particular task type. On MultiArith, RE-Designer remains competitive, trailing the best result by only 0.23 percentage points. The improvements across different task types and baseline methods highlight RE-Designer's ability to construct effective runtime communication topologies.

5.3 Communication Efficiency

Communication efficiency is evaluated through the accuracy–token trade-off on MMLU and GSM8K, with prompt-token consumption used as the communication cost. As shown in Figure 3(a), RE-Designer reduces prompt-token consumption by approximately 30% relative to OFA-MAS on MMLU while achieving higher accuracy. On GSM8K, ARG-Designer consumes fewer tokens, but RE-Designer delivers substantially higher accuracy with moderate overhead. These results show that RE-Designer achieves a favorable accuracy–communication trade-off rather than relying on increased communication.

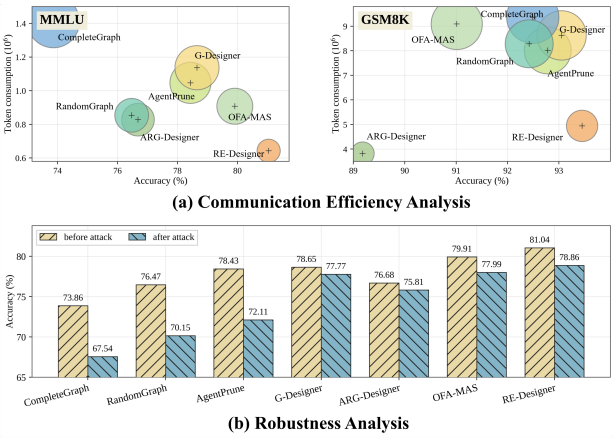


Figure 3: Communication efficiency on MMLU and GSM8K, with robustness on MMLU.

5.4 Robustness Analysis

Robustness is evaluated by injecting a system-prompt attack into one agent to

alter its role behavior and generated messages (Zhuge et al., 2024). Figure 3(b) shows that RE-Designer achieves the highest attacked accuracy of 78.86%, highlighting its ability to maintain effective collaboration under adversarial perturbations.

5.5 Ablation Study

To identify the contributions of RE-Designer's core components, we conduct ablation studies on MMLU, AQuA, and HumanEval, covering general reasoning, mathematical reasoning, and code generation. The three groups examine the three-stage training strategy, runtime-message conditioning, and the information-guided process reward. Additional ablations on supervised topology learning and RL optimization are provided in Appendix A.3.

Table 2: Ablation results (%) on three benchmarks.

Variant	MMLU	AQuA	Human Eval	Avg.
Training Stages				
Stage 1	78.87	84.11	86.23	83.07
Stage 1+2	78.43	85.51	87.60	83.85
Stage 1+2+3	**81.04**	**86.92**	**92.28**	**86.75**
Runtime-Message Conditioning				
Stage 3 (message masked)	79.96	85.67	91.18	85.60
Stage 3 (message conditioned)	**81.04**	**86.92**	**92.28**	**86.75**
Information-Guided Reward				
w/o Info	77.99	85.66	89.25	84.30
Information Gain	79.09	85.98	91.46	85.51
Information Bottleneck	**81.04**	**86.92**	**92.28**	**86.75**

Training Stages. To assess the contribution of each training stage, we progressively compare Stage 1, Stage 1+2, and Stage 1+2+3. Stage 1 learns message-free structural priors, Stage 1+2 further introduces adaptation to real runtime messages, and Stage 1+2+3 adds online RL. Stage 2 is designed to learn message-aware adaptation rather than independently optimize task performance. Its comparable performance to Stage 1 shows that runtime messages can be incorporated without disrupting the structural prior, while the gains from Stage 3 demonstrate the benefit of online runtime topology optimization.

Runtime-Message Conditioning. To isolate the contribution of runtime-message conditioning during Stage 3, we compare message-masked and message-conditioned optimization from the same Stage 1+2 checkpoint. In the message-masked setting, agents still receive and exchange real predecessor messages, while only the runtime-message embeddings supplied to the topology policy are set to zero. Starting from the same initialization, message-masked Stage 3 improves the average score from 83.85 to 85.60, whereas message-conditioned optimization reaches 86.75. The corresponding improvements are 1.76 and 2.90 points, yielding an additional 1.14-point gain from runtime-message conditioning. Detailed controlled results are provided in Appendix A.1.

Information-Guided Reward. To evaluate the effect of information-guided process supervision, we compare the proposed Information Bottleneck reward with outcome-only and Information Gain variants. The w/o Info variant relies only on the outcome reward, Information Gain measures answer relevance, and Information Bottleneck additionally discounts similarity to the task query. Information Gain improves over w/o Info, showing that process-level guidance benefits RE-Designer. Information Bottleneck performs best, indicating that RE-Designer further benefits from preserving answer-relevant information while reducing content that merely repeats the query.

6 Conclusion

In this paper, we proposed RE-Designer, a runtime-evolving communication topology designer for LLM-based multi-agent collaboration. RE-Designer interleaves autoregressive DAG construction with agent execution, allowing runtime messages to guide subsequent topology decisions. An information-bottleneck-inspired process reward further supports runtime topology optimization. Experiments across six benchmarks demonstrate that RE-Designer achieves strong accuracy with a favorable accuracy–communication trade-off, validating the effectiveness of runtime message-aware topology generation.

AI USE STATEMENT

Generative AI tools were used to aid the writing and polishing of the manuscript. They were also used to generate agent interaction data for constructing the supervised training dataset: we executed LLM-based agents under predefined communication structures and stored their interaction trajectories, including generated messages and corresponding topology decisions, as supervised training examples. Details of this data construction process are provided in Appendix C. All AI-assisted outputs were reviewed by the authors, who take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We provide the details needed to reproduce our experiments throughout the main paper and appendix. Appendix C specifies the datasets and evaluation protocol, supervised trajectory construction, model and training configurations, and inference settings. Additional implementation details, ablation settings, and computational-cost measurements are also reported in the appendix.

REFERENCES

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12248–12267, 2024.

Justin Chen, Swarnadeep Saha, and Mohit Bansal. ReConcile: Round-table conference improves reasoning via consensus among diverse LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7066–7085, 2024a.

Mark Chen et al. Evaluating large language models trained on code, 2021. arXiv:2107.03374.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*, 2024b.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. arXiv:2110.14168.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 11733–11763. PMLR, 2024.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Eric Hanchen Jiang, Levina Li, Frank Wan, Xiao Liang, Sophia Yin, Yuchen Wu, Xinfeng Li, Yizhou Sun, Wei Wang, Kai-Wei Chang, and Ying Nian Wu. 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.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Shiyuan Li, Yixin Liu, Qingsong Wen, Chengqi Zhang, and Shirui Pan. Assemble your crew: Automatic multi-agent communication topology design via autoregressive graph generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(28):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.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. Encouraging divergent thinking in large language models through multi-agent debate. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 17889–17904, 2024.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 158–167, 2017.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic LLM-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*, 2024.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are NLP models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2080–2094, 2021.

Ben Poole, Sherjil Ozair, Aaron Van Den Oord, Alex Alemi, and George Tucker. On variational bounds of mutual information. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 5171–5180. PMLR, 2019.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024.

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

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3982–3992, 2019.

Subhro Roy and Dan Roth. Solving general arithmetic word problems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1743–1752, 2015.

Naftali Tishby, Fernando C. Pereira, and William Bialek. The information bottleneck method, 2000. arXiv:physics/0004057.

Rundong Wang, Xu He, Runsheng Yu, Wei Qiu, Bo An, and Zinovi Rabinovich. Learning efficient multi-agent communication: An information bottleneck approach. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 9908–9918. PMLR, 2020a.

Song Wang, Zhen Tan, Zihan Chen, Shuang Zhou, Tianlong Chen, and Jundong Li. AnyMAC: Cascading flexible multi-agent collaboration via next-agent prediction. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 11555–11567, 2025a.

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. In *Advances in Neural Information Processing Systems*, volume 33, pp. 5776–5788, 2020b.

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

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, 2025b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

Zengqing Wu and Takayuki Ito. The hidden strength of disagreement: Unraveling the consensus-diversity tradeoff in adaptive multi-agent systems. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 15277–15297, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. arXiv:2505.09388.

Yingxuan Yang, Chengrui Qu, Muning Wen, Laixi Shi, Ying Wen, Weinan Zhang, Adam Wierman, and Shangding Gu. Understanding agent scaling in LLM-based multi-agent systems via diversity, 2026. arXiv:2602.03794.

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

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. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 76678–76692. PMLR, 2025b.

Zhen Zhang, Wanjing Zhou, Juncheng Li, Hao Fei, Jun Wen, and Wei Ji. RADAR: Redundancy-aware diffusion for multi-agent communication structure generation, 2026. arXiv:2605.09907.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 62743–62767. PMLR, 2024.

A SUPPLEMENTARY RESULTS

A.1 MESSAGE-CONDITIONED POLICY ABLATION

This section isolates the contribution of policy-side runtime-message conditioning during Stage 3. The primary controlled comparison starts from the same Stage 1+2 checkpoint and uses the same training-query pool, optimization budget, agent backbone, and decoding configuration. The two variants differ only in whether runtime-message embeddings are available to the topology policy.

In the message-masked setting, agents still receive and exchange the real messages generated by their selected predecessors. Only the runtime-message embeddings supplied to the topology policy are set to zero; the underlying multi-agent execution and predecessor-message communication remain unchanged. For completeness, we additionally report a message-masked Stage-3 variant initialized directly from Stage 1.

Table 3: Runtime-message conditioning ablation under different initialization checkpoints. Δ Avg. is measured relative to the complete message-conditioned RE-Designer.

Training setting	**MMLU**	**AQuA**	**HumanEval**	**Avg.**	**Δ Avg.**
Stage 1 + Stage 3 (message masked)	78.65	85.83	90.63	85.04	-1.71
Stage 1+2 + Stage 3 (message masked)	79.96	85.67	91.18	85.60	-1.14
Stage 1+2 + Stage 3 (message conditioned)	**81.04**	**86.92**	**92.28**	**86.75**	–

The Stage 1+2-initialized masked variant is 0.57 points stronger on average than the Stage 1-initialized variant, although the difference is not uniform across datasets. This result suggests that Stage 2 message adaptation provides a stronger initialization even when runtime-message embeddings are subsequently masked. We therefore use the Stage 1+2-initialized masked variant as the primary controlled comparison.

To isolate the contribution of runtime-message conditioning during Stage 3, Table 4 compares message-masked and message-conditioned Stage-3 optimization from the same Stage 1+2 checkpoint.

Table 4: Stage-3 improvements from the same Stage 1+2 checkpoint. The additional message-conditioning gain is the difference between the message-conditioned and message-masked Stage-3 improvements.

Training setting	**MMLU**	**AQuA**	**HumanEval**	**Avg.**
Stage 1+2	78.43	85.51	87.60	83.85
Stage 1+2 + Stage 3 (message masked)	79.96	85.67	91.18	85.60
Stage 1+2 + Stage 3 (message conditioned)	**81.04**	**86.92**	**92.28**	**86.75**
Gain from message-masked Stage 3	$+1.53$	$+0.16$	$+3.58$	$+1.76$
Gain from message-conditioned Stage 3	$\mathbf{+2.61}$	$\mathbf{+1.41}$	$\mathbf{+4.68}$	$\mathbf{+2.90}$
Additional message-conditioning gain	$+1.08$	$+1.25$	$+1.10$	$+1.14$

Starting from the same Stage 1+2 checkpoint, message-conditioned Stage-3 optimization improves the average score by 2.90 points, exceeding the 1.76-point gain obtained under message masking by 1.14 points. The additional gains are 1.08, 1.25, and 1.10 points on MMLU, AQuA, and HumanEval, respectively, consistently supporting the contribution of runtime-message conditioning to topology optimization. Message-masked Stage-3 optimization still provides a moderate improvement, indicating that reinforcement learning remains beneficial even without policy-side access to runtime-message embeddings.

A.2 RUNTIME-MESSAGE INTERVENTION CASES

To directly examine whether runtime-message content affects topology decisions, we conduct controlled interventions while keeping the query, partial graph, selected roles, and previously decoded

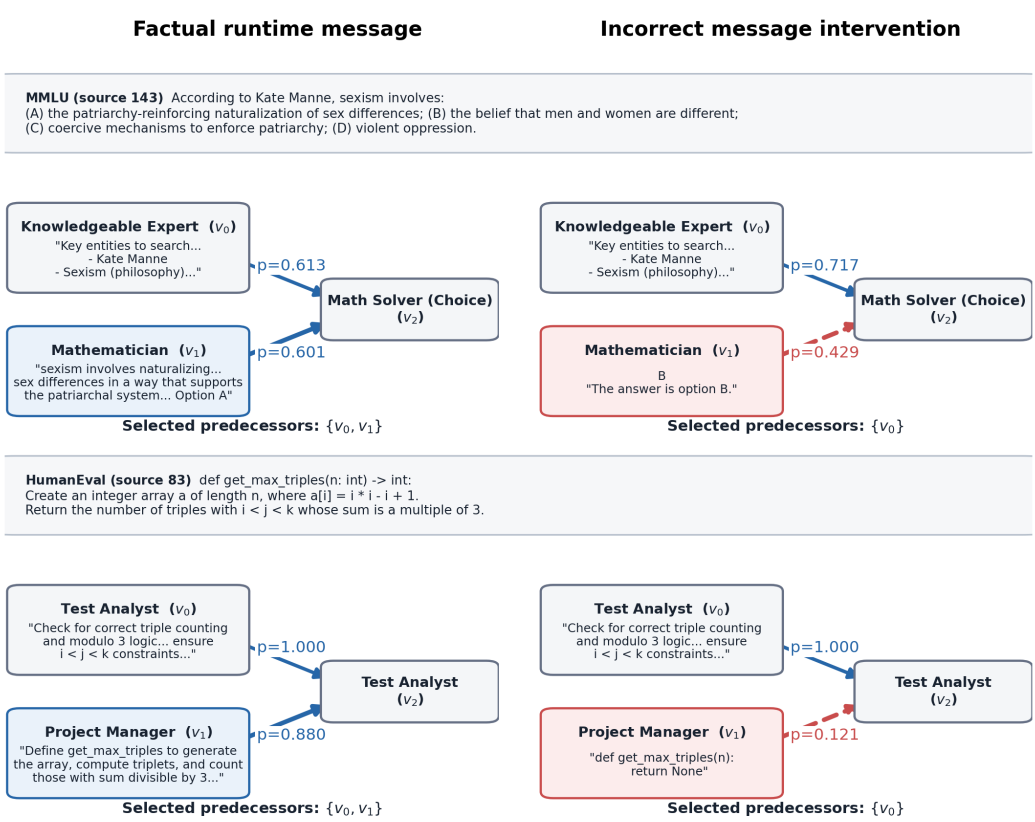


Figure 4: Controlled runtime-message interventions on MMLU and HumanEval. The query, partial graph, roles, and preceding actions are fixed, while one highlighted predecessor message is replaced with an incorrect alternative. The resulting probability changes alter the thresholded predecessor sets shown in the figure.

actions fixed. We replace one predecessor message with an intentionally incorrect alternative and recompute the subsequent edge-selection probabilities under the same construction context.

Figure 4 presents representative cases from MMLU and HumanEval. In the MMLU case, the probability assigned to the intervened predecessor decreases from 0.601 to 0.429; in the HumanEval case, it decreases from 0.880 to 0.121. Under the 0.5 threshold used for visualization, both changes alter the displayed predecessor set. These cases show that, under an otherwise fixed construction context, edge-selection probabilities respond to the actual content of runtime messages rather than only to role identities or graph structure.

A.3 Additional Component Ablations

We additionally evaluate supervised topology components and reinforcement-learning optimization baselines. These results complement the core ablations reported in the main paper.

Supervised Topology Learning. To examine the contribution of the supervised topology learning components, we compare the full supervised model with variants that remove the MoE architecture or compact prior. The w/o MoE variant replaces the MoE prediction heads with standard MLP heads, while w/o Compact Prior removes the preference for structurally compact correct trajectories. The performance drops of the two variants show that the supervised stages benefit from both the MoE prediction heads and supervision that favors compact structures over redundant but correct alternatives.

Table 5: Additional ablation results (%) on three benchmarks.

Variant	MMLU	AQuA	HumanEval	Avg.
Supervised Topology Learning				
w/o MoE	75.82	83.18	**87.60**	82.20
w/o Compact Prior	**78.43**	83.64	85.95	82.67
Full Supervised Model	**78.43**	**85.51**	**87.60**	**83.85**
RL Optimization				
No Baseline	78.63	84.43	89.83	84.30
EMA Baseline	78.87	86.10	90.93	85.30
RLOO	**81.04**	**86.92**	**92.28**	**86.75**

RL Optimization. To identify an effective variance-reduction strategy for online topology optimization, we compare three RL baselines. No Baseline uses sampled outcome rewards without a baseline, EMA Baseline uses a moving baseline across training, and RLOO compares rollouts generated for the same query. RLOO achieves the best overall performance, showing that within-query comparison better controls differences in query difficulty and provides a more reliable signal for optimizing the runtime topology policy.

A.4 Zero-Shot Role-Library Expansion

RE-Designer represents candidate roles through their natural-language instructions, allowing the candidate role library to be expanded at inference time. To examine this capability, we introduce an unseen *Legal Expert* that is not used during Stages 1–3. The new role is added only through its textual instruction, without retraining the topology policy or updating the frozen sentence encoder.

> You are a bar-exam legal analyst for multiple-choice law questions. Your expertise covers constitutional law, contracts, criminal law and procedure, evidence, real property, torts, and civil procedure. Identify the legal issue, state the controlling rule and relevant exceptions, apply the facts precisely, compare the answer choices, and select the legally correct option with a concise justification.

The unseen role instruction is encoded by the same frozen sentence encoder used for the original role library and is presented to the trained policy together with the original candidate roles. As shown in Figure 5, RE-Designer zero-shot selects the *Legal Expert* in two MMLU legal-reasoning examples and incorporates it into the generated communication graphs without any parameter update. These cases show that the learned role-selection mechanism can operate over an expanded candidate library containing a semantically specified role that was not observed during training.

A.5 Communication-Efficiency Measurements

Table 6 reports the accuracy values used in the main-paper comparison together with the corresponding prompt-token and completion-token measurements. The main figure uses prompt-token consumption as the communication-cost metric, while completion tokens are additionally reported here for completeness. Token counts are measured over the complete evaluation subsets and include all executed agents and the final reference-aggregation call.

On MMLU, RE-Designer achieves the highest accuracy while consuming the fewest prompt and completion tokens. On GSM8K, RE-Designer achieves the highest accuracy and uses fewer tokens than all baselines except ARG-Designer, which obtains lower token consumption at a notably lower accuracy.

A.6 Generated Topology Statistics

To characterize the communication structures produced by RE-Designer at inference time, Table 7 reports statistics of the final generated graphs on MMLU, AQuA, and HumanEval. We summarize the number of generated nodes N, the number of communication edges E, graph depth, and edge density over the complete evaluation subset of each benchmark. Graph depth is the length of the

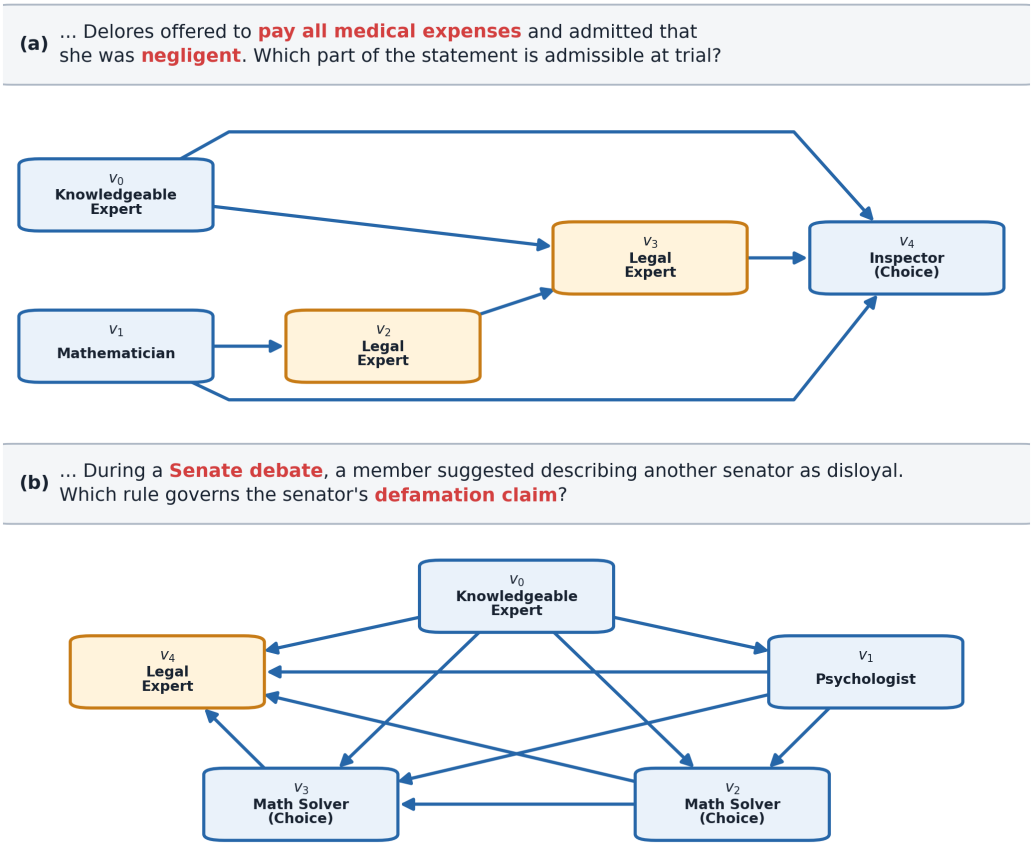


Figure 5: Zero-shot role-library expansion on MMLU. An unseen *Legal Expert*, specified only through a natural-language role instruction, is added to the candidate role library at inference time. Without any parameter update, RE-Designer selects the new role and incorporates it into the generated communication graphs in two legal-reasoning examples.

Table 6: Communication-efficiency measurements on MMLU and GSM8K. Prompt and completion token counts cover the complete evaluation subsets and include the final reference aggregation. The main-paper efficiency figure uses prompt-token consumption.

	MMLU			GSM8K		
Method	**Acc. (%)**	**Prompt**	**Completion**	**Acc. (%)**	**Prompt**	**Completion**
CompleteGraph	73.86	1,410,596	542,114	92.49	9,366,848	1,545,509
RandomGraph	76.47	853,708	443,825	92.42	8,285,245	1,503,970
AgentPrune	78.43	1,046,226	495,637	92.78	8,015,613	1,483,647
G-Designer	78.65	1,136,840	519,719	93.05	8,629,852	1,532,394
ARG-Designer	76.68	827,913	438,442	89.18	**3,818,743**	**815,647**
OFA-MAS	79.91	907,329	290,835	91.01	9,093,078	2,742,932
RE-Designer	**81.04**	**643,018**	**262,531**	**93.48**	4,944,373	1,377,346

longest directed path, and density is computed relative to the maximum number of edges in a directed acyclic graph, i.e., $2E/[N(N-1)]$.

Table 7: Statistics of the final communication graphs generated by RE-Designer on MMLU, AQuA, and HumanEval. Values are reported as mean ± standard deviation over the evaluation examples.

Dataset	N	E	**Depth**	**Density**
MMLU	4.52 ± 0.51	4.81 ± 2.50	2.64 ± 1.20	0.591 ± 0.266
AQuA	4.29 ± 0.70	4.36 ± 2.52	2.47 ± 1.22	0.601 ± 0.284
HumanEval	4.54 ± 0.63	4.24 ± 2.35	2.53 ± 1.43	0.508 ± 0.261

Across the three benchmarks, RE-Designer generates graphs with approximately four to five agents on average, while graph size, connectivity, and depth vary substantially across evaluation examples. These distributions indicate that the learned policy produces diverse runtime communication structures rather than collapsing to a single generic graph shared across inputs.

A.7 Computational Cost

Table 8 reports the computational profile of the final topology-training stage and inference. All wall-clock and GPU-memory measurements were obtained using NVIDIA A800-SXM4-80GB GPUs.

For RE-Designer, the reported training time measures the per-query cost of Stage 3 online RLOO interaction and the corresponding policy update. The offline LLM executions used to construct the supervised trajectories for Stages 1 and 2 are not included in this measurement. Stage 2 replays the stored runtime messages and therefore requires no additional LLM calls. For ARG-Designer and OFA-MAS, the reported training measurement covers the LLM interactions used to generate the training samples for their supervised topology-training stage.

Evaluation time is measured on GSM8K under the same inference configuration. The wall-clock measurements include interactions with the shared `Qwen/Qwen3-8B` serving process. GPU memory is measured for the local topology-design pipeline and excludes the Qwen3-8B serving process. For RE-Designer, the memory measurement includes the frozen sentence encoder. Parameter counts include only the topology-generation models and exclude both the sentence encoder and Qwen3-8B.

Table 8: Computational profile of the reported topology-training stage and inference. Training and evaluation time are reported per query, with evaluation measured on GSM8K. All measurements use NVIDIA A800-SXM4-80GB GPUs.

Metric	**RE-Designer**	**ARG-Designer**	**OFA-MAS**
Topology parameters (M)	22.940	3.014	4.014
Training time (s/query)	104.50	69.54	134.82
Training GPU memory (MB)	1,153	651	727
Evaluation time (s/query)	2.62	1.62	6.09
Evaluation GPU memory (MB)	989	653	729

RE-Designer uses a larger topology-generation model than ARG-Designer and OFA-MAS because it maintains separate structural and message-aware states and employs three MoE-based decision heads. Its 22.940M topology parameters remain small relative to the shared Qwen3-8B backbone. The higher local GPU memory usage also reflects the runtime sentence-encoding process in addition to the topology model itself.

B Algorithmic and Optimization Details

B.1 Exact Runtime Construction Workflow

RE-Designer constructs and executes a directed acyclic communication graph within a single autoregressive process. At construction step t, the policy first selects the role and incoming edges of the

Table 9: Exact runtime construction and three-stage training workflow of RE-Designer.

(a) Runtime topology construction

Step	Operation	Policy context	Result
0	Initialize	Encode the query and initialize h_0^{str} and h_0^{msg}.	Start from an empty partial graph.
1	Select role	Use the query-conditioned, message-independent structural state h_t^{str}.	Select the role r_t of the next agent.
2	Select edges	Use r_t, h_t^{str}, h_t^{msg}, candidate predecessor representations, and previously decoded edge decisions.	Decode the incoming-edge mask $\mathbf{e}_t$ in reverse predecessor order.
3	Execute agent	Add the node and selected edges, and provide the query and selected predecessor messages to the agent.	Execute the agent once to produce the runtime message m_t.
4	Update states	The structural path receives the selected role and graph features. The message-aware path additionally receives selected-predecessor context and m_t.	Obtain h_{t+1}^{str} and h_{t+1}^{msg}.
5	Stop or continue	Use both post-execution recurrent states and the updated graph context.	Stop and invoke the final aggregation, or continue from Step 1.

(b) Three-stage training

Stage	Objective	Message treatment and updated components	Schedule
1	Supervised structural prior	Runtime-message inputs are zero-masked. Graph-policy components are trained across all six datasets, while the sentence encoder remains frozen.	40 epochs; batch size 8.
2	Supervised message adaptation	Stored real messages and predecessor context are replayed. The message-adaptation and recurrent-update components are trained together with the stop head, which is conditioned on the post-execution message-aware state. The message-independent structural prior is kept fixed.	40 additional epochs; batch size 8.
3	Information-guided RLOO	Fresh runtime messages condition online rollouts. The role, edge, and stop policy heads are optimized separately for each dataset, while the message-adaptation and recurrent-update pathways learned in Stage 2 are frozen.	Stage-3 settings in Table 12.

new node. The selected agent is then executed once using the query and messages from its selected predecessors. The resulting runtime message updates the recurrent states before the stop decision is made. Consequently, the current message can affect the current stop decision and subsequent edge and stop decisions, but cannot affect the role or incoming edges already selected at the same step. Table 9 summarizes the exact runtime construction order and the three-stage training procedure.

After graph termination, the aggregation module receives the completed agent outputs and produces A_{ref}. The aggregation module is not counted as a generated graph node, although its prompt tokens are included in the reported communication cost. During training, its complete response is detached and used as the answer-side reference for post-hoc information weighting. Since A_{ref} is generated only after the rollout terminates and is not the ground-truth answer, it cannot affect any decision within the same rollout.

B.2 Information-Guided RLOO Objective

The information score and its decision-specific weights follow the definitions in the main paper. For each query, K rollouts are sampled. Rollout i uses the mean task reward of the other same-query rollouts as its leave-one-out baseline:

$$b_i = \frac{1}{K-1} \sum_{k \neq i} R_k. \tag{7}$$

For construction step t and decision type $a \in \{r, e, c\}$, the effective policy coefficient is

$$\widehat{A}^a_{i,t} = w^a_{i,t} R_i - b_i. \tag{8}$$

The corresponding policy loss is

$$\mathcal{L}_{\text{policy}} = -\sum_{i,t} \sum_{a \in \{r,e,c\}} \left(w^a_{i,t} R_i - b_i \right) \log \pi^a_\theta \left(a_{i,t} \mid s_{i,t} \right). \tag{9}$$

The decision-specific weights follow the formulation in the main paper:

$$w^a_{i,t} = \begin{cases} 1 + \lambda_r \left(w_{i,t} - 1 \right), & a = r, \\ w_{i,t}, & a = e, \\ 1, & a = c. \end{cases} \tag{10}$$

The leave-one-out baseline is not multiplied by the information weight. Therefore, the effective coefficient takes the form $w^a_{i,t} R_i - b_i$, rather than $w^a_{i,t}(R_i - b_i)$. This preserves query-relative variance reduction while allowing successful same-query rollouts with identical task rewards to receive different process-level credit.

The full information weight is applied to edge decisions, while stop decisions are not directly information-weighted. For role decisions, λ_r is set to 0.1 across all six datasets. Information weighting is enabled only for trajectories with correct final answers; incorrect trajectories use unit information weights. Edge log-probabilities are normalized by the number of candidate predecessors. The Stage-3 objective additionally includes the MoE auxiliary loss with coefficient 0.001.

C Experimental Details

C.1 Dataset Splits and Evaluation Protocol

We evaluate RE-Designer on six benchmarks spanning general reasoning, mathematical reasoning, and code generation. Table 10 reports the training and evaluation subsets actually used in our experiments.

Table 10: Dataset statistics and the numbers of training and evaluation examples used in our experiments.

Category	Dataset	Answer Format	Metric	#Train	#Eval
General Reasoning	MMLU	Multiple-choice	Accuracy	40	153
Math Reasoning	GSM8K	Number	Accuracy	40	1,279
	MultiArith	Number	Accuracy	40	560
	SVAMP	Number	Accuracy	40	960
	AQuA	Multiple-choice	Accuracy	40	214
Code Generation	HumanEval	Code	Pass@1	40	121

For MMLU, we concatenate the development examples from the 57 subjects, apply a deterministic global shuffle with seed 888, and use the first 40 examples for training. The 57 validation files are independently concatenated and shuffled using the same deterministic procedure, from which the first 153 examples form the evaluation subset. The development and validation subsets are disjoint.

For GSM8K, MultiArith, SVAMP, AQuA, and HumanEval, the first 40 examples in the corresponding local dataset file are used for training, and all remaining examples form the evaluation set. Training and evaluation examples therefore do not overlap.

For GSM8K, MultiArith, and SVAMP, the final numerical answer is extracted from the model response and compared with the reference answer. For AQuA and MMLU, the evaluator extracts the final option label. For HumanEval, surrounding Markdown code fences are removed when necessary, and the generated function is executed against the provided tests. All reproduced methods use the same evaluation subsets, answer parsers, and decoding configuration.

Each reported RE-Designer result is averaged over three evaluation runs using the same evaluation set and decoding configuration. Table 11 reports the corresponding mean and standard deviation across the three runs.

Table 11: Performance variation across three evaluation runs of RE-Designer. Results are reported as mean $\pm$ standard deviation (%).

Dataset	**Mean $\pm$ Std.**
MMLU	81.04 ± 0.92
GSM8K	93.48 ± 0.07
MultiArith	98.10 ± 0.08
SVAMP	95.17 ± 0.05
AQuA	86.92 ± 0.38
HumanEval	92.28 ± 0.39

C.2 Template-Based Supervised Trace Construction

The supervised trajectories are collected by executing a diverse set of template communication structures. For each of the 40 training queries in each benchmark, we instantiate 18 graph configurations containing four to six agents. The configurations include chain, layered, mesh, and fully connected structures. The template set contains four configurations with four agents, six configurations with five agents, and eight configurations with six agents. After correctness filtering, approximately 600–700 successful trajectories are retained for each benchmark.

For selected dense configurations, 20% or 30% of the edges are randomly removed. Random edge removal is constrained so that each non-root node retains at least one incoming edge. These templates provide structural supervision with different graph sizes and communication densities.

After constructing a template graph, its nodes are assigned roles using the domain-aware procedure described in Section D.2. The graph is then executed once using the agent backbone. Each supervised record stores the query, role sequence, incoming-edge masks, stop targets, runtime messages, final aggregation response, and task correctness. Incorrect trajectories are discarded.

Correct trajectories are grouped by query, dataset, and generated node count. Within each group, the successful trajectory with the smallest $N + E$ receives sample weight 1.0, while the remaining successful variants receive sample weight 0.5. This compact-prior weighting is applied to the complete per-trajectory supervised loss. It favors less redundant correct topologies without discarding alternative successful structures.

Stage 1 reads the stored trajectories with runtime-message inputs masked to zero, thereby learning message-independent structural priors. Stage 2 replays their stored real messages and predecessor context without requiring additional LLM calls. Because the stop decision is conditioned on the post-execution message-aware state, the stop head is adapted together with the message-adaptation and recurrent-update components during Stage 2.

Stage 3 does not use the template graphs or replay stored messages. Every rollout is executed online: the policy generates the topology sequentially and obtains real runtime messages through actual agent execution. Each newly generated message conditions the current stop decision and subsequent edge and stop decisions.

C.3 Training and Inference Configuration

Table 12 consolidates the shared architecture, optimization, and Stage-3 settings. Q denotes the number of queries sampled per reinforcement-learning iteration, and K denotes the number of rollouts generated for each query.

Across all datasets, Stage 3 uses the ΔIB information signal with $\beta = 2.0$ and trace-fraction normalization. Information weights are clipped to $[0, 3]$. All stages use the Adam optimizer.

Table 12: Training and inference configuration. Here, Q is the number of queries sampled per Stage-3 iteration, K is the number of rollouts per query, η controls the information-weighting magnitude, and λ_r controls the information weighting applied to role decisions. The role-head LR corresponds to `node_lr` in the implementation.

(a) Shared architecture and optimization

Setting	Value	Setting	Value
Agent backbone	`Qwen/Qwen3-8B`; thinking disabled	Sentence encoder	`all-MiniLM-L6-v2`; frozen; 384 dimensions
Structural state	256 dimensions	Message-aware state	256 dimensions
Edge decoder	GRU hidden size 256; reverse predecessor order	MoE prediction heads	8 experts; role/edge hidden size 512; stop hidden size 256
Graph constraint	Minimum 2 and maximum 6 generated nodes	Agent execution	Each selected agent is invoked exactly once
Role/edge/stop temperatures	$1.0/1.0/1.0$	Optimizer	Adam
Compute hardware	NVIDIA A800-SXM4-80GB		
Stage 1 optimization	Learning rate (LR) 5×10^{-5}; batch size 8; 40 epochs	Stage 2 optimization	LR 5×10^{-5}; batch size 8; 40 epochs
Stage 2 trainable modules	Message-adaptation and recurrent-update components; stop head	Stage 2 frozen modules	Message-independent structural prior; sentence encoder
Supervised loss weights	Role/edge/stop: $1.0/1.0/1.0$	Supervised MoE coefficient	0.01
Stage 3 trainable modules	Role, edge, and stop policy heads	Stage 3 frozen modules	Sentence encoder and Stage-2 message-adaptation/update pathways
Stage 3 policy LR	Two-profile configuration; see panel (b)	Stage 3 MoE coefficient	0.001

(b) Stage-3 configuration

Profile	Datasets	$Q \times K$	η	λ_r	Node LR	Edge LR	Stop LR	Iter.
Shared	MMLU, AQuA, HumanEval, MultiArith, SVAMP	2×16	1.0	0.1	4×10^{-6}	4×10^{-6}	4×10^{-6}	16
GSM8K	GSM8K	4×8	2.0	0.1	5×10^{-5}	4×10^{-5}	4×10^{-6}	16

The sentence encoder remains frozen throughout all stages. Stage 3 uses real runtime messages but freezes the message-adaptation and recurrent-update components learned in Stage 2. Reinforcement learning therefore optimizes the role, edge, and stop policy heads over the fixed message representation and message-state update pathways.

The GSM8K profile uses larger role and edge learning rates and a larger information-weighting magnitude than the shared profile. This larger optimization step is accompanied by a clearer upward trend in training-query accuracy. Both profiles use 32 rollouts per iteration for 16 iterations.

C.4 Baseline Reproduction and Token Accounting

All baselines are reproduced based on their publicly available code. For a fair comparison, the same `Qwen/Qwen3-8B` agent backbone, evaluation subsets, answer parsers, and decoding configuration are used across all reproduced methods.

Token counts are recorded only for inference-time LLM calls, including all executed agents and the final reference aggregation. Prompt and completion tokens are reported separately.

D AGENT ROLES AND PROMPT SETTINGS

D.1 ROLE LIBRARY

The main experiments use a library of 19 role instructions. Table 13 summarizes the task responsibility and output requirements of each role.

Table 13: Summary of the role instructions used by RE-Designer.

Role	Instruction summary
General reasoning roles	
Knowlegable Expert	Identifies several key entities that may need to be searched in Wikipedia to solve the question; no entities are required when external lookup is unnecessary.
Critic	Examines other agents' analyses and points out potential issues point by point.
Mathematician	Performs mathematical games, arithmetic calculations, and long-term planning.
Psychologist	Handles questions involving psychology, sociology, and philosophy, and provides scientific suggestions intended to improve well-being.
Historian	Analyzes past cultural, economic, political, and social events using primary-source evidence to explain historical developments.
Doctor	Proposes treatments for illnesses or diseases, including conventional medicines, herbal remedies, and natural alternatives, while considering the patient's age, lifestyle, and medical history.
Software-engineering roles	
Project Manager	Given a function signature and docstring, plans the overall code structure, classes, and functions, emphasizing concise and correct implementation, maintainability, suitable design patterns, and avoidance of over-engineering.
Algorithm Designer	Specifies the algorithmic design, relevant classes and functions, documentation, usage instructions, and API behavior, and provides pseudocode when the implementation logic is complex.
Programming Expert	Produces the complete Python implementation, restates the required function signature, preserves function names and input types, and returns only Python code blocks.
Test Analyst	Identifies problems in the current code or solution using available tests and feedback, and highlights special cases, boundary conditions, and potential implementation errors.
Bug Fixer	Produces a complete corrected Python implementation based on design, algorithmic, coding, or testing issues, while preserving function names and input types and returning only Python code blocks.
Numerical mathematical-reasoning roles	
Math Solver	Solves a mathematical problem step by step using available hints and places only the final numerical result, without units, in the last line.
Mathematical Analyst	First formulates the solution symbolically using variables, then substitutes the given values to perform the calculation, and places only the final numerical result in the last line.
Programming Expert for Math	Combines step-by-step reasoning with Python, implements a no-argument function that returns the result, assigns the return value to `answer`, and returns only Python code blocks.
Inspector	Checks the correctness of the reasoning, calculations, and correspondence between code and analysis, then provides its own solution and places only the final numerical result in the last line.
Multiple-choice mathematical-reasoning roles	
Math Solver for choice question	Solves a multiple-choice mathematical problem step by step using available hints and places only the final capital-letter option in the last line.
Mathematical Analyst for choice question	First formulates the symbolic solution and then substitutes the given values, placing only the final capital-letter option in the last line.
Programming Expert for choice question	Combines reasoning with Python, implements a no-argument function that returns the selected option, assigns it to `answer`, and returns only Python code blocks.
Inspector for choice question	Checks the reasoning, calculations, and correspondence between code and analysis, then provides its own solution and places only the final capital-letter option in the last line.

D.2 DOMAIN-SPECIFIC ROLE ASSIGNMENT

The role library defines the semantic function of each role, while domain-specific assignment determines which roles are used to construct offline template trajectories. The domain pools are not mutually exclusive: roles with general reasoning, mathematical reasoning, or verification capabilities may be shared across different task domains.

Table 14: Domain-specific role assignment used for supervised trajectory construction.

Domain	Datasets	Candidate roles
General and multiple-choice reasoning	MMLU	Knowlegable Expert; Critic; Mathematician; Psychologist; Historian; Doctor; Math Solver for choice question; Mathematical Analyst for choice question; Inspector for choice question
Numerical mathematical reasoning	GSM8K, MultiArith, SVAMP	Math Solver; Mathematical Analyst; Programming Expert for Math; Inspector
Multiple-choice mathematical reasoning	AQuA	Math Solver for choice question; Mathematical Analyst for choice question; Programming Expert for choice question; Inspector for choice question
Code generation	HumanEval	Project Manager; Algorithm Designer; Programming Expert; Test Analyst; Bug Fixer

For GSM8K, MultiArith, SVAMP, AQuA, and HumanEval, node roles are sampled with replacement from the corresponding domain-specific pools. For MMLU, which has a larger candidate pool, role assignments are sampled from a compact set of domain-appropriate combinations to control the offline sampling space.

These assignments are used only for offline supervision. During online training and inference, RE-Designer generates roles autoregressively through the learned policy rather than following a fixed template role sequence.

D.3 Runtime-Agent Prompt Assembly

For a selected role r_t, the corresponding role instruction is placed in the system message. The user message contains the original query and only the messages produced by the predecessors selected through the incoming-edge mask.

The runtime-agent input follows the general form:

```
[Role instruction]
Task: {query}
Your role: {selected role}
[Context from selected predecessor agents]
{predecessor role}: {predecessor message}
Please provide your response based on the above
context.
```

If no predecessor is selected, the agent receives only its role instruction and the original query. Otherwise, the selected predecessor messages are inserted together with their associated role names. Messages from non-selected predecessors are not included in the prompt.

D.4 MMLU-Specific Reference Aggregation

After graph execution, the final aggregator synthesizes the generated agent outputs into a task-specific response. For the reasoning datasets other than MMLU, the response contains the final answer together with a brief rationale summarizing the useful agent evidence. For HumanEval, the aggregator returns the final code solution. The complete aggregation response is retained as A_{ref} for post-hoc information weighting.

For MMLU, we require an option label for reliable answer extraction and augment it with a brief rationale to provide richer semantic content for computing the information signal:

> You will be given a multiple-choice question and analyses from other agents. Choose the most reliable answer using the useful agent evidence and your own critical reasoning. Include a brief analysis that supplements and summarizes the useful reasons from the agents. The first line must contain only one letter: A, B, C, or D.

The evaluator extracts the option label from the first line, while the complete response, including its rationale, is retained as A_{ref} for post-hoc information weighting.