DON'T JUST OPTIMIZE PROMPTS: EVOLVING AGENT COMMUNICATION PROTOCOLS WITH EVOPROTOCOL

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent systems (MAS) tackle complex tasks by coordinating role-specialized agents with focused context. By iteratively optimizing and evolving their components from execution feedback, these systems can improve their performance and efficiency over time. However, prior work mainly focuses on optimizing agent prompts or coordination workflows, while the inter-agent communication protocol, a fundamental component that determines what agents communicate and how they exchange information, remains largely underexplored. To address this limitation, we propose EvoProtocol, a framework that exposes the communication protocol as an explicit optimization variable and evolves it jointly with agent prompts. On MAS-PromptBench, jointly evolving prompts and protocols outperforms prompt optimization under every fixed protocol on all five tasks, even when compared against the best-performing fixed protocol for each task. On BrowseComp-Plus with a dynamic planner–executor workflow, EvoProtocol improves accuracy by 5.5–16 points over prompt optimization with fixed protocols, and its cost-aware variant achieves higher accuracy at lower cost than prompt optimization with the hand-designed PACT protocol. These results show that co-evolving communication protocols with agent prompts yields better combinations of prompts and protocols than prompt optimization under fixed protocols.

1 INTRODUCTION

Multi-agent systems (Guo et al., 2024; Tran et al., 2025; Wei et al., 2026; Ning et al., 2026) provide a promising approach to solving long-horizon tasks by distributing work across role-specific agents, each operating with its own context. Self-evolution adapts such systems to individual tasks by revising their components based on feedback from past executions, as the optimal design of a multi-agent system can vary across tasks (Fang et al., 2025; Gao et al., 2025). A prominent line of work performs such adaptation through LLM-driven optimization, which iteratively revises components of individual agents, such as agent prompts, while leaving the mechanisms of inter-agent communication largely underexplored (Khattab et al., 2023; Yuksekgonul et al., 2024; Agrawal et al., 2026; Chu et al., 2026).

Beyond agent prompts, a multi-agent system also includes the inter-agent communication protocol, a critical component that determines how messages are composed and what information produced by earlier agents reaches subsequent agents. However, existing work targeting inter-agent communication (Ma et al., 2026; Tieu et al., 2026; Huang et al., 2026) treats this protocol as a fixed part of the harness, the execution framework that runs the agents, rather than as an optimization target. This fixing limits what self-evolution can reach. As shown in Figure 1, no fixed protocol consistently achieves the best performance across tasks, even after prompt optimization. A self-evolving multi-agent system should therefore optimize the communication protocol jointly with the agent prompts.

To realize this co-optimization, we propose EvoProtocol, which exposes the inter-agent communication protocol to self-evolution alongside the agent prompts. Starting from the harness-default prompts and protocols, EvoProtocol repeatedly executes the multi-agent system, constructs feedback from the resulting execution traces, and uses an LLM optimizer to revise both the prompts and the protocols.

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.

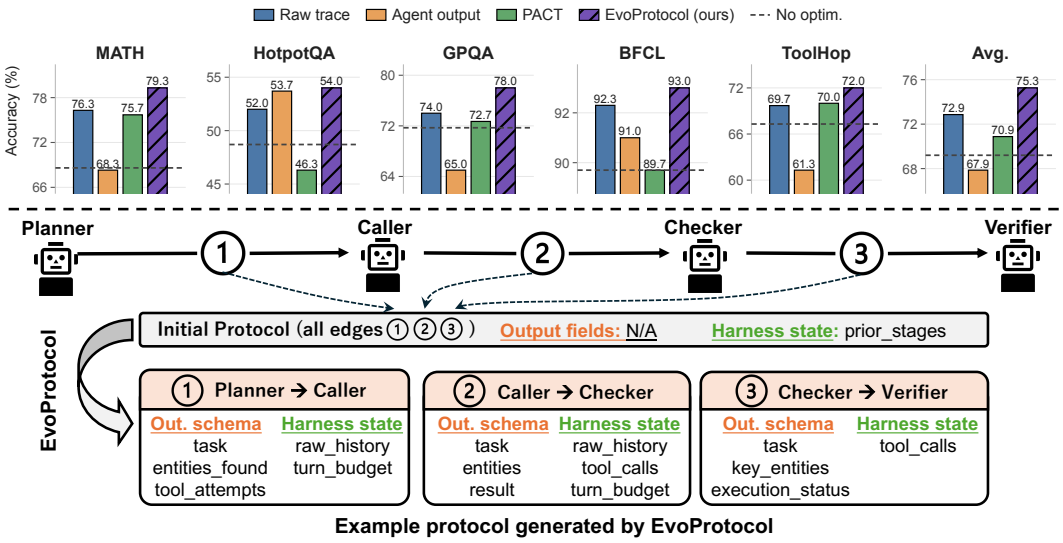

Figure 1: (Upper) Accuracy on MAS-PromptBench with prompt optimization under fixed protocols; the dashed line denotes the unoptimized system. The best fixed protocol varies across tasks, and none reaches EvoProtocol. (Lower) Protocols evolved by EvoProtocol on ToolHop (harness state defined in Table 6).

Jointly optimizing prompts and protocols requires two components that existing self-evolution methods lack. First, the optimizer must be able to rewrite the protocol, yet the protocol is typically built into the harness rather than exposed as a separate object. We therefore represent each protocol as text, placing prompts and protocols in a single search space so that the LLM optimizer can revise them jointly. Specifically, a protocol specifies the content delivered to the target agent, which consists of the structured output of the source agent and information selected from what the harness accumulates during execution. These two sources correspond to the two main forms of communication in existing harnesses (CrewAI Inc., 2026; LangChain Inc., 2026; Wu et al., 2023). Second, the optimizer needs feedback on how each protocol affects the target agent, whereas task-level metrics such as accuracy reflect only the aggregate effect of many communication events. We therefore segment the execution trace at each communication point and use the outcomes of the target agent's subsequent actions, such as tool-call success or failure, as local feedback for the protocol.

We evaluate our approach on two representative benchmarks for static and dynamic multi-agent workflows, MAS-PromptBench (Bai & Shi, 2026) and BrowseComp-Plus (Chen et al., 2025), and compare it with prompt optimization under fixed protocols. Starting from the same default prompts and protocol, EvoProtocol improves average accuracy on MAS-PromptBench by 6.1 points, whereas prompt-only optimization improves it by only 3.7 points. On BrowseComp-Plus, EvoProtocol improves accuracy by 16.0 and 5.5 points over prompt optimization with the harness-default and PACT (Huang et al., 2026) protocols, respectively, and its cost-aware variant reduces execution cost by 30% while maintaining accuracy. Our analysis further suggests that these gains require joint optimization: the evolved protocol either provides a foundation on which the optimized prompts work effectively, or allows the optimizer to discover prompts that prompt optimization does not find.

In summary, our contributions are as follows:

- Through quantitative analysis, we identify the inter-agent communication protocol as a distinct optimization target beyond agent prompts within a self-evolution loop.
- We propose EvoProtocol, which formulates protocols into a searchable space and introduces local feedback that attributes execution outcomes to individual protocols, enabling their joint optimization.
- Across static and dynamic multi-agent workflows, extensive experiments demonstrate that EvoProtocol consistently outperforms optimizing prompts alone under the fixed protocols.

2 Background and Related Work

2.1 Multi-agent systems

We adopt the formalization of multi-agent systems introduced by Bai & Shi (2026). A multi-agent system is a tuple $M = (A, G, P)$, where $A = \{A_0, \ldots, A_n\}$ is a set of LLM agents, G is a coordination workflow, and P is the communication protocol. Each agent $A_i = (\theta_i, \pi_i)$ consists of an LLM with weights θ_i and an agent prompt π_i. The collection of agent prompts is denoted as $\pi = (\pi_0, \ldots, \pi_n)$. Throughout this work, LLM weights are treated as fixed parameters.

We represent the coordination workflow as a directed graph $G = (A, E)$, where each edge $e = (A_s, A_t) \in E$ indicates that a source agent A_s can send messages to a target agent A_t during execution. The communication protocol is defined per edge as $P = \{P_e\}_{e \in E}$, where P_e specifies how the message from A_s is delivered to A_t. When A_t is invoked, this message forms part of the context it receives.

Formally, the full context c_t of A_t consists of:

$$c_t = (\pi_t, \sigma_t, r_t), \tag{1}$$

where π_t is the agent prompt, σ_t is configuration context determined by the harness, an execution framework that runs the systems, and r_t is runtime context produced during execution. Given c_t, A_t generates actions (e.g., tool calls) with its LLM θ_t over multiple steps, and the generated content and resulting observations accumulate in r_t. Its output is then either passed to subsequent agents or returned as the system's final output. Thus, the execution trace τ of a multi-agent system is the collection of contexts c_t of all invoked agents during a single run.

2.2 Self-evolving multi-agent systems

Self-evolution adapts a multi-agent system by revising its components based on feedback from past executions (Fang et al., 2025; Gao et al., 2025). Let $\mathcal{T}$ denote a distribution over task inputs x and evaluation targets m, and let μ be a task metric that scores the system output $\mathcal{M}(x; \pi, G, P)$ against m. Given a set of components $S \subseteq \{\pi, G, P\}$ exposed to optimization, self-evolution solves

$$\max_{S} \ \mathbb{E}_{(x,m)\sim\mathcal{T}}\big[\mu\big(\mathcal{M}(x; \pi, G, P),\ m\big)\big], \qquad S \subseteq \{\pi, G, P\}, \tag{2}$$

where only the components indexed by S are updated. This paper considers offline evolution, where pre-collected datasets are used for iterative refinement.

Existing methods can be categorized into three groups based on the choice of S. Prompt optimization takes $S = \{\pi\}$, keeping G and P fixed (Zhou et al., 2022; Pryzant et al., 2023; Khattab et al., 2023; Yang et al., 2024; Opsahl-Ong et al., 2024; Yuksekgonul et al., 2024; Agrawal et al., 2026; Chu et al., 2026). Topology optimization takes $S = \{G\}$ or $S = \{\pi, G\}$, exposing the coordination workflow to search (Zhang et al., 2024; Zhuge et al., 2024; Zhang et al., 2025). Neither approach considers P as an explicit search space. Program search, in contrast, does not fix S explicitly: it edits the program as a whole and thereby modifies π, G, and P jointly (Hu et al., 2025). Although this indirectly exposes P to search, it does not optimize P explicitly.

2.3 Inter-agent communication

In practice, P is specified by the harness, and existing harnesses realize it in substantially different ways: Some deliver only the output content generated by the source agent (CrewAI Inc., 2026), whereas others forward content accumulated by the harness during execution, such as the source agent's execution history (LangChain Inc., 2026; Wu et al., 2023).

Recently, a few studies have begun to focus on inter-agent communication itself, either by designing a generic protocol or by instructing agents what to communicate. PACT introduces a generic protocol that adds structured action, state, and result fields to the shared history forwarded between agents (Huang et al., 2026). CPE evolves an agent prompt that decides whether the agent communicates in free text or through a structured UI, in settings where the target is either a user or another agent (Ma et al., 2026). ReActNet introduces a controller LLM that specifies, for each edge, what the source agent should convey to the target agent, injecting the instruction at run time rather than

fixing it in the agent prompt (Tieu et al., 2026). However, these methods do not optimize P. Hand-designed protocols fix P manually, and instruction-based methods change what agents are instructed to communicate while leaving P at the harness-default.

Two empirical findings further suggest that fixing P limits what self-evolution can achieve. First, the form of P determines how much prompt optimization can improve a system. MAS-PromptBench optimizes π under different types of P and reports that the improvements from prompt optimization grow with the structure of P (Bai & Shi, 2026). Second, no single choice of P is best across settings. The best-performing design depends on the task and the agent configuration (Huang et al., 2026). Together, these findings imply that fixing P in advance constrains the outcome of self-evolution. To verify this directly, we optimize π under each of three fixed protocols on MAS-PromptBench with the same optimizer and budget: the *raw trace*, which forwards the outputs of all prior stages as accumulated by the harness; the *agent output*, which passes only the output generated by the source agent; and PACT. As shown in Figure 1, the best-performing P still depends on the task even after prompt optimization, and some choices of P leave the prompt-optimized system below the unoptimized one with the harness-default protocol. In Section 3.1, we therefore expose P to self-evolution alongside π by proposing EvoProtocol.

3 EvoProtocol: A Framework for Self-Evolving Protocol

This section introduces EvoProtocol, a self-evolution framework that jointly optimizes the agent prompts π and the communication protocol P in a textual space, to maximize the task metric μ. EvoProtocol is orthogonal to the choice of the underlying optimizer, and can therefore be combined with LLM-based textual optimizers in principle. Figure 2 provides an overview of the self-evolving loop. In each iteration, the multi-agent system with the current combination (π, P) is executed on a minibatch of tasks, producing execution traces τ and task metric μ. For each optimization variable, an LLM optimizer generates a new candidate using the feedback texts generated from these traces and metric values. The candidate system is then evaluated and accepted only if its task metric μ improves.

To realize this loop, we introduce two components: a textual parameterization of P in Section 3.1 and a feedback construction that attributes execution outcomes to individual protocols in Section 3.2. Finally, Section 3.3 instantiates the framework on GEPA (Agrawal et al., 2026), a reflective prompt optimizer.

3.1 Textual Parameterization for Communication Protocol

Given a coordination workflow G, EvoProtocol exposes both the agent prompts π and the communication protocol P to self-evolution. Following Equation (2), this optimization problem can be formulated as:

$$(\pi^*, P^*) = \arg\max_{\pi, P} \mathbb{E}_{(x,m)\sim\mathcal{T}} \left[\mu\big(\mathcal{M}(x; \pi, G, P), m\big)\right]. \tag{3}$$

The prompts π are already text, whereas the protocol is generally implemented inside the harness and differs across harnesses. EvoProtocol places the protocol P in the same textual space, so that the LLM optimizer can revise them jointly. Since P is defined per edge, we parameterize each P_e separately.

Following existing harnesses (Section 2.3), we compose a message on each edge from two sources: the output generated by the source agent A_s and the information accumulated by the harness. For the first, we define an output schema F_e that declares the named fields that A_s should produce. For the second, we introduce the harness state $\mathcal{H}$, which is a fixed set of the content accumulated by the harness, and the optimizer selects the forwarded harness state $h_e \subseteq \mathcal{H}$ for each edge. The harness state $\mathcal{H}$ can include the agent history and action history accumulated by the harness. Its concrete contents depend on the harness used for each benchmark, and we list the available harness states for each setting in Appendix A. Then, the harness composes the message that is delivered to A_t from these two sources. We represent this composition as a message template T_e, which specifies how the fields of F_e and the forwarded state harness state h_e are combined into the delivered message. Thus, the protocol P_e consists of

$$P_e = (T_e, F_e, h_e). \tag{4}$$

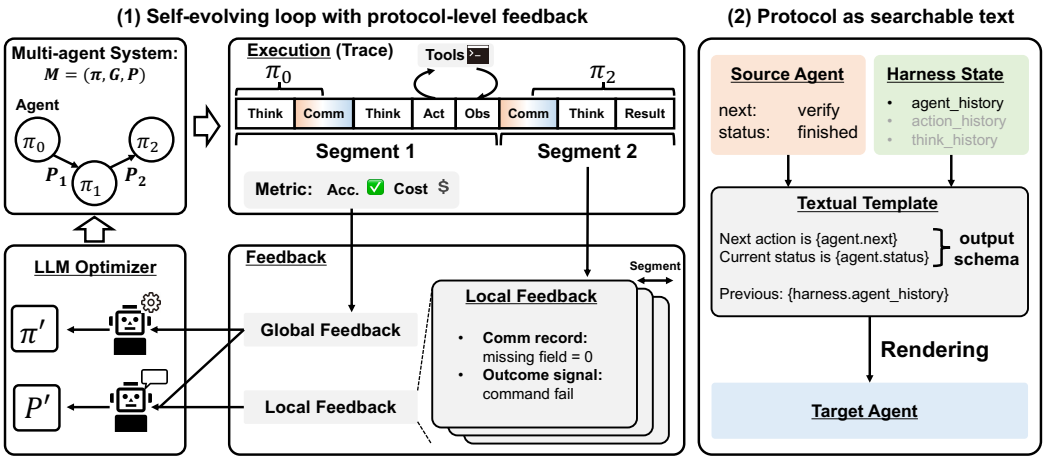

Figure 2: Overview of EvoProtocol. The left figure shows our self-evolving loop, and the right figure shows our textual parameterization for the communication protocol.

Figure 2 (right) shows a concrete example of P_e.

The fields of F_e and the elements of h_e appear in T_e as named placeholders, so both sources of message content are expressed within a textual template. Because F_e, h_e, and T_e are all expressed as text, the protocol can be searched using the same class of textual optimization mechanisms used for agent prompts.

To ensure that the source agent A_s follows the output schema F_e, the harness enforces it in two ways. First, F_e is exposed to the source agent as part of its configuration context σ_s as defined in Equation (1). This configuration preserves the distinction in Section 2 as π_s specifies the agent prompt, whereas F_e is an edge-specific interface configuration provided by the harness through σ_s. Second, the harness validates the produced output fields against F_e and discards any field not declared in F_e. Each field in F_e is treated as optional: a missing field does not invalidate the output, and T_e renders it as an empty value.

Extension to Multiple Protocols A single protocol uses the same communication format for every message sent along an edge. However, in practice, an edge is traversed many times within an execution, and the purpose of communication may differ across traversals, so a single protocol is not necessarily the best for all of them. We support this variation by allowing multiple protocols on each edge. This extension differs from topology optimization, because G specifies only whether two agents can communicate, whereas multiple protocols specify how they communicate.

When multiple protocols are available on an edge, the source agent selects which protocol to use for each communication. We also leverage the configuration context σ_s to realize this routing in protocols. Specifically, for each protocol on the edge, the harness appends its name, the fields of its output schema, and a routing description to the context σ_s. These texts are also part of the search space of P. The source agent selects a protocol by emitting its name as part of its output, so a missing or invalid selection falls back to the harness-default protocol and is recorded in the trace.

3.2 Protocol Feedback from Execution Traces

The only signal directly observable from a protocol is whether the source agent emits the fields declared in F_e, which does not indicate whether the delivered information is useful to the target agent. The effectiveness of a message becomes observable only through the subsequent behavior of the target agent, whereas the task metric μ reflects only the aggregate effect of all messages in a trace. To attribute outcomes to individual protocols, we split the trace τ into segments at communication points (Figure 2).

Each segment starts with a message and contains the subsequent actions and observations. When several messages are delivered with no action in between, their outcomes cannot be separated, so

they are grouped into the same segment. For each segment, we construct a local feedback signal that consists of a communication record and an outcome signal. The communication record lists, for each message in the segment, which fields of F_e the source agent populated and which it left empty, together with the selected protocol when the edge carries multiple protocols. The outcome signal summarizes the results of the actions in the segment with rule-based signals, such as whether each action succeeds or fails; the concrete signals are task-specific and listed in Appendix B. The local signal is attributed to every edge along which a message was sent in the segment, and μ serves as a global feedback signal for all protocols.

3.3 Instantiation with GEPA

We instantiate EvoProtocol on GEPA (Agrawal et al., 2026), a reflective prompt optimizer that maintains a Pareto-optimal candidate set and iteratively refines it through execution, reflection, proposal, and acceptance. We register each agent prompt π_i and each protocol P_e as independent modules, and extend GEPA to handle protocols as follows:

- **Initialization.** Each P_e is initialized by mapping the harness-default protocol onto (T_e, F_e, h_e), so that the seed reproduces the default behavior.
- **Reflection.** Prompts receive task-specific feedback constructed from τ and μ, shared across all methods, whereas protocols receive the local and global signals in Section 3.2. The concrete signals for both are listed in Appendix B.
- **Proposal.** The LLM jointly rewrites F_e and T_e, and h_e is set to the harness-state elements referenced by placeholders in T_e. A candidate is rejected before execution if any placeholder in T_e matches neither a field in F_e nor an element of $\mathcal{H}$. With multiple protocols on an edge, the LLM instead receives the current protocols with their usage counts and proposes communication patterns absent from the set; protocols never selected in the minibatch are pruned, except for the first one, which is retained as the fallback.
- **Acceptance.** Following GEPA, a candidate is accepted if it improves the task metric μ over its parent. For cost-aware optimization, we extend μ to include the measured USD cost, and regard a candidate as improving μ if it either improves accuracy or reduces cost while retaining at least 80% of its parent's accuracy.

4 Experiments

4.1 Experimental Setup

Baselines. We use GEPA (Agrawal et al., 2026) as the optimization framework, and compare EvoProtocol with prompt optimization under fixed protocols. We consider two fixed protocols that correspond to the two standard forms of communication in existing harnesses (Section 2.3): the *raw trace*, which forwards the outputs of all prior stages as accumulated by the harness, and the *agent output*, which passes only the output generated by the source agent. On MAS-PromptBench, we use the raw trace (the harness default), the agent output, and PACT (Huang et al., 2026) as the fixed protocols; on BrowseComp-Plus, we use the agent output (the harness default) and PACT. In contrast, EvoProtocol jointly optimizes the agent prompts and communication protocol. We exclude CPE (Ma et al., 2026) and ReActNet (Tieu et al., 2026) from our evaluation. Both adjust the content of communication through agent instructions while keeping the protocol P fixed to the harness default, and the effect of such prompt-level adaptation is covered by the prompt optimization baselines.

Optimization setup. All methods, including the fixed-protocol baselines, are optimized with GEPA using the same budget, optimizer LLM, and hyperparameters; they differ only in the set of optimized variables. Following GEPA, the optimization budget is defined as the number of executions, where one execution corresponds to running the multi-agent system on one example once, including both training and validation executions. In each iteration, GEPA updates all optimized modules concurrently, i.e., all agent prompts for the baselines and all agent prompts and protocols for EvoProtocol. Within each task, all methods use the same feedback function for updating agent prompts. The final candidate is selected based on the optimization data alone, and the test examples

Table 1: Results on MAS-PromptBench with a four-agent sequential workflow. All optimized methods are instantiated with GEPA under the same optimization budget. Prompt-only baselines optimize π while keeping P fixed, whereas EvoProtocol jointly optimizes π and P. We report average accuracy over three evaluation seeds.

Method	Protocol	Optimized variables	MATH	HotpotQA	GPQA	BFCL	ToolHop	Avg.
No optim.	Raw Trace	–	68.6	48.7	71.7	89.7	67.3	69.2
GEPA	Raw Trace	π	<u>76.3</u>	52.0	74.0	<u>92.3</u>	69.7	<u>72.9</u>
GEPA	Agent output	π	68.3	<u>53.7</u>	65.0	91.0	61.3	67.9
GEPA	PACT	π	75.7	46.3	<u>72.7</u>	89.7	<u>70.0</u>	70.9
EvoProtocol	Optimized	π, P	**79.3**	**54.0**	**78.0**	**93.0**	**72.0**	**75.3**

are never used during optimization. For cost-aware optimization, we apply the acceptance criterion in Section 3.3 and select the lowest-cost candidate that satisfies the accuracy constraint on the optimization data. The concrete harness state and rule-based feedback signals depend on the benchmark harness and task. We provide their exact definitions for each setting in Appendix A and Appendix B.

MAS-PromptBench. For static coordination workflow, we evaluate 5 benchmarks from MAS-PromptBench (Bai & Shi, 2026). This evaluation includes MATH, GPQA, HotpotQA, ToolHop, and BFCL. These benchmarks are built mainly on a LangChain-based harness (LangChain Inc., 2026), which uses raw trace as the default. Thus, we use this protocol as the starting point of EvoProtocol. Each task is solved by four agents arranged in a sequential workflow, where each agent is invoked once and sends a message to the next agent. Following the original setup of the benchmark, each agent uses Qwen3.5-9B, and we perform optimization once and report accuracy averaged over three evaluation seeds. The optimization data and budget depend on the task. For GPQA, HotpotQA, and ToolHop, we use 50 examples each for training and validation, sample a minibatch of 25 training examples in each update, and set the optimization budget to 1,000 executions. For MATH and BFCL, we use 25 examples each for training and validation, a minibatch of 8, and a budget of 250 executions. We evaluate on the test set of the original benchmark, whose size varies across tasks. We use Qwen3.5-122B-A10B-FP8 as the LLM optimizer.

BrowseComp-Plus. For a dynamic workflow, we evaluate a planner–executor (PE) workflow on BrowseComp-Plus (Chen et al., 2025), where the planner invokes the executor dynamically during execution. We use its standard evaluation harness and implement executor invocation by extending Claude Code's PE workflow, which passes only the source agent's output between the planner and the executor and thus serves as the harness-default protocol. Unlike the sequential workflow in MAS-PromptBench, the planner may invoke the executor multiple times within a task. We therefore evaluate both a single protocol and multiple protocols on our framework. We use GPT-5.6 Sol (medium) as the planner and GPT-5.6 Luna (high) as the executor (OpenAI, 2026). We perform optimization once and report accuracy averaged over two seeds. We randomly sample 25 examples from BrowseComp-Plus for optimization and 100 disjoint examples for evaluation. In each update, GEPA executes each example twice to account for the stochasticity of the workflow. We use all 25 optimization examples rather than a minibatch because a single task on this benchmark takes a long time to complete; evaluating all tasks at once maximizes parallelism and thus reduces the wall-clock time of the search. We use GPT-5.6 Sol (high) as the LLM optimizer to update the system. We set the GEPA optimization budget to 600 executions.

4.2 Results on MAS-PromptBench

Table 1 shows that EvoProtocol outperforms prompt optimization under every fixed protocol across all tasks. Under fixed protocols, the gains of prompt optimization depend on which protocol is fixed, and the best choice differs across tasks. By exposing the protocol to optimization, EvoProtocol removes this dependence and finds systems that prompt-only optimization under any of the fixed protocols does not.

Table 2: Results on BrowseComp-Plus under accuracy-only optimization. The fixed-protocol baselines optimize only agent prompts, while EvoProtocol jointly optimizes agent prompts and the communication protocol. Results are averaged over two evaluation seeds on 100 held-out examples. Wrong and timeout (T/O) are aggregated over 200 executions. Timeout is set to three hours for evaluation. $/roll. represents the average cost per execution. The pricing table used for cost calculation is provided in Table 7.

Method	Protocol P	Optimized variables	Acc.	Wrong	T/O	$/roll.
GEPA	Agent output	π	62.0	76	**0**	**0.172**
GEPA	PACT	π	72.5	43	12	0.555
EvoProtocol (single)	Optimized	π, P	**78.0**	**21**	23	0.796
EvoProtocol (multiple)	Optimized	π, P	77.5	32	13	0.672

Table 3: Cost decomposition on BrowseComp-Plus. We compare accuracy-only and cost-aware optimization and break down execution cost into planner and executor token usage. Token counts are reported per execution.

			Outcome		Planner Usage			Executor Usage			USD	
Method	Protocol P	Obj.	Acc.	$/roll	in	cach.	out	in	cach.	out	Planner	Executor
GEPA	Agent output	accuracy	62.0	0.172	6k	21k	1.1k	252k	127k	35.9k	0.076	0.096
		+ cost	62.5	0.251	6k	28k	1.9k	467k	738k	35.1k	0.101	0.150
		Δ	+0.5	+46%	±0%	+29%	+69%	+86%	+482%	−2%	+33%	+57%
GEPA	PACT	accuracy	72.5	0.555	16k	46k	5.5k	799k	3,355k	50.0k	0.268	0.287
		+ cost	59.0	0.239	11k	26k	2.6k	242k	191k	32.3k	0.148	0.091
		Δ	−13.5	−57%	−31%	−43%	−53%	−70%	−94%	−35%	−45%	−68%
EvoProtocol	Optimized	accuracy	77.5	0.672	6k	26k	1.3k	1,708k	8,312k	67.6k	0.083	0.589
		+ cost	78.5	0.467	7k	22k	1.2k	1,197k	4,128k	51.8k	0.083	0.384
		Δ	+1.0	−30%	+18%	−12%	−10%	−30%	−50%	−23%	−0%	−35%

4.3 Results on BrowseComp-Plus

Task performance. Table 2 shows that EvoProtocol also improves performance in the dynamic PE workflow. Using the agent-output protocol, prompt optimization achieves an accuracy of 62.0%, and adopting the hand-designed PACT protocol increases accuracy to 72.5%. The single-protocol variant of EvoProtocol outperforms the agent-output and PACT baselines by 16.0 and 5.5 points, respectively. The improvement comes mainly from fewer wrong answers: the number of wrong answers decreases from 76 and 43 for the two baselines to 21 with a single protocol. These results demonstrate that protocol optimization remains effective even when the sequence of communications over G is determined dynamically at execution time.

Effect of multiple protocols. Using multiple protocols gives similar accuracy to using a single optimized protocol, but reduces execution cost. The single protocol achieves 78.0% accuracy at a cost of $0.80 per execution, while multiple protocols achieve 77.5% accuracy at $0.67 per execution. This suggests that using different protocols across executor invocations mainly affects how much computation is used, rather than further improving task accuracy. In particular, the return protocol remains unchanged from its initialization, while the planner-to-executor protocols are substantially revised (Table 11). This indicates that adapting the planner-to-executor communication can be sufficient to preserve performance without requiring a richer return protocol, which in turn reduces the number of input tokens processed by the planner.

However, the optimized protocols still incur higher execution cost than the fixed-protocol baselines. This motivates us to explicitly include execution cost in protocol optimization. We adopt the multiple-protocol variant for this experiment because it reduces cost compared with the single protocol.

Cost-aware Optimization. We next optimize both protocols and prompts with execution cost included in the objective. For cost-aware optimization, we use the acceptance criterion in Section 3.3 and select the lowest-cost candidate whose accuracy on the optimization data is at least 80% of the best candidate's accuracy. As shown in Table 3, cost-aware optimization reduces the average execution cost of EvoProtocol from $0.67 to $0.47 while maintaining accuracy (77.5% to 78.5%).

(a) Protocol ablation.

Method	Protocol	MATH	HotpotQA	GPQA	BFCL	ToolHop
No optim.	Raw Trace	68.6	48.7	71.7	89.7	67.3
GEPA	Raw Trace	76.3	52.0	74.0	92.3	69.7
EvoProtocol	Raw Trace	80.3	44.0	78.0	93.3	66.3
EvoProtocol	Optimized	79.3	54.0	78.0	93.0	72.0

(b) Joint vs. sequential optimization.

Method	Optimized	Avg.
No optim.	–	69.2
GEPA	π	72.9
Sequential	$\pi \rightarrow P$	71.3
Joint	π, P	75.3

Table 4: Ablation studies on MAS-PromptBench. Accuracy is averaged over three different seeds. (a) Protocol ablation. Agent prompts are fixed to those obtained by EvoProtocol, and only the protocol is replaced with the raw trace. (b) Joint versus sequential optimization. Sequential optimization first optimizes the agent prompts and then optimizes only the protocol under the same total budget as joint optimization.

The fixed-protocol baselines show different behavior. For the harness-default protocol, GEPA accepted only one candidate during cost-aware optimization, and its cost is higher than that of accuracy-only optimization ($0.17 to $0.25), while accuracy changes little. PACT reduces the cost from $\$0.56$ to $\$0.24$, but accuracy drops from 72.5% to 59.0%. In contrast, EvoProtocol reduces execution cost without compromising task accuracy. As a result, cost-aware EvoProtocol outperforms accuracy-only prompt optimization with PACT in both accuracy and cost (78.5% at $0.47 vs. 72.5% at $0.56).

4.4 Ablation Study

Protocol ablation. We first study how the optimized protocol contributes to performance on MAS-PromptBench. We fix the agent prompts obtained by EvoProtocol and replace only the protocol: we replace the optimized protocol with the raw trace. Table 4(a) shows that protocols contribute to the self-evolution in two different ways. On HotpotQA and ToolHop, accuracy drops below that of the unoptimized system, indicating that the optimized prompts rely on the optimized protocol. On MATH, GPQA, and BFCL, the optimized prompts still outperform prompt-only optimization, indicating that exposing the protocol leads the optimizer to prompts that prompt-only optimization does not find under the same budget. This suggests that the gains of EvoProtocol arise not from the protocol alone but from joint optimization, which distributes improvements between prompts and protocols.

Effect of joint optimization . We next examine whether prompts and protocols need to be optimized jointly. We compare EvoProtocol on MAS-PromptBench with sequential optimization, which first optimizes the agent prompts under the raw trace and then optimizes the protocol with the prompts fixed. To align the total optimization budget, we allocate half of the executions to prompt optimization and the remaining half of the executions to protocol optimization. As shown in Table 4(b), joint optimization achieves better performance on average across five tasks compared with sequential optimization, suggesting that prompts optimized under the raw trace constrain what the subsequent protocol optimization can achieve. Together with Table 4(a), this result indicates that prompts and protocols need to be adapted together during optimization: optimizing the protocol for prompts fixed in advance does not recover the gains of joint optimization.

5 Conclusion

We presented EvoProtocol, a self-evolution framework that treats the inter-agent communication protocol as an explicit optimization variable and evolves it jointly with agent prompts. To realize this, we introduced a textual parameterization of protocols and a segment-level feedback signal that attributes execution outcomes to individual protocols. Across MAS-PromptBench and BrowseComp-Plus, joint optimization outperforms prompt optimization under every fixed protocol, and its cost-aware variant reduces execution cost while maintaining accuracy. More broadly, our results suggest that the components of a multi-agent system are not independent design choices: what self-evolution can discover is bounded by which components are exposed to search, and exposing them jointly allows self-evolution to discover systems that optimizing any subset in isolation does not. We hope this perspective encourages further work on component-level co-evolution, in which each component of a multi-agent system is exposed as an explicit optimization variable and all exposed components are evolved jointly.

AI USE STATEMENT

We used generative AI tools to assist with the following: implementing parts of the evaluation scripts, organizing and formatting experimental results into tables, drafting and polishing the manuscript, checking notation and technical details, and improving the visual presentation of figures. The authors reviewed all AI-assisted outputs, including testing the code, checking the manuscript, and verifying all reported results against the raw experimental logs. The authors take full responsibility for the content of this paper.

ETHICS STATEMENT

This work does not involve human subjects, personally identifiable information, or the collection of new data; all experiments use publicly available benchmarks under their respective licenses.

REPRODUCIBILITY STATEMENT

We describe EvoProtocol in Section 3.1 and the experimental setup, including models, data splits, optimization budgets, and hyperparameters, in Section 4. All methods share the same optimizer, budget, and hyperparameters and differ only in the set of optimized variables. The appendix provides the harness state, the feedback signals for protocol optimization, the pricing used for cost calculation, and the learned protocols.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Juyang Bai and Laixi Shi. Mas-promptbench: When does prompt optimization improve multi-agent llm systems? *arXiv preprint arXiv:2606.23664*, 2026.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, et al. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*, 2025.

Jaewon Chu, Jinwoo Seo, Jaewon Cho, Jeehye Na, Yunyang Xiong, Youngdae Kim, and Hyunwoo J Kim. Agentgrad: Intervention-guided prompt optimization for multi agent systems. *arXiv preprint arXiv:2609.08572*, 2026.

CrewAI Inc. Crewai: Framework for orchestrating role-playing autonomous ai agents. `https://github.com/crewaiinc/crewai`, 2026. Accessed: 2026-09-15.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

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

Chen Huang, Yuhao Wu, and Wenxuan Zhang. What should agents say? action-state communication for efficient multi-agent systems. *arXiv preprint arXiv:2606.05304*, 2026.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

LangChain Inc. Langgraph: Build resilient agents. `https://github.com/langchain-ai/langgraph`, 2026. Accessed: 2026-09-15.

Xinbei Ma, Jiyang Qiu, Yao Yao, Zheng Wu, Yijie Lu, Xiangmou Qu, Jiaxin Yin, Xingyu Lou, Jun Wang, Weiwen Liu, et al. Communication policy evolution for proactive llm agents. *arXiv preprint arXiv:2606.14314*, 2026.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

OpenAI. Gpt-5.6: Frontier intelligence that scales with your ambition. `https://openai.com/index/gpt-5-6/`, 2026. Accessed: 2026-09-21.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 7957–7968, 2023.

Katherine Tieu, Dongqi Fu, Yinglong Xia, Hong Li, Hong Yan, and Jingrui He. Inference-time graph engineering for multi-agent llm workflows. *arXiv preprint arXiv:2609.05774*, 2026.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*, 2025.

Tianxin Wei, Ting-Wei Li, Zhining Liu, Xuying Ning, Ze Yang, Jiaru Zou, Zhichen Zeng, Ruizhong Qiu, Xiao Lin, Dongqi Fu, et al. Agentic reasoning for large language models. *arXiv preprint arXiv:2601.12538*, 2026.

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.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

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, 2025.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.

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

Table 5: Harness state available for Browse-Comp Plus.

Name	Description
`original_question`	The original user question for the whole run.
`remaining_budget`	Tool calls left in the run's shared tool-call budget.
`executor_turn_ceiling`	The executor's turn limit for the current subtask.
`wall_clock_remaining_sec`	Seconds left before the run's wall-clock deadline.
`prior_findings`	Goals and answers of earlier subtasks on this question, oldest first.
`prior_docids`	Deduplicated ids of every document consulted in earlier subtasks.
`attempted_queries`	Distinct search queries issued so far, in issue order.
`search_and_get_document_calls`	Tool calls spent so far, as separate search and get_document counts.
`tool_call_parse_retries`	Executor turns whose first parse failed and were re-prompted.
`no_tool_answer_retries`	Executor turns that attempted an answer before any tool call.
`route_retry_count`	Model-API route retries needed so far.
`planner_parse_retries`	Planner turns whose first parse failed and were re-prompted.
`findings_entries_dropped`	Findings-ledger entries dropped from rendering so far.
`prior_documents`	Every document surfaced by a search so far, as "[docid] title", opened or not.
`raw_trace`	Executor events of all completed subtasks, one line per event.
`planner_trace`	The planner's own turns so far, one line per turn.
`unread_documents`	Documents surfaced by a search but never opened.
`repeat_unread_documents`	Unread documents surfaced by two or more distinct searches.
`read_fraction`	Opened documents over surfaced documents, as an integer percentage.
`zero_yield_searches`	Search calls that surfaced no previously unseen document.
`query_yield`	One line per search call: "⟨query⟩ → ⟨n⟩ new", the number of unseen documents it surfaced.

Table 6: Harness state available for MAS-PromptBench.

Name	Description
`prior_stages`	Every prior stage's output, exactly as the runner hands it over.
`agent_outputs`	Each prior agent's own text, with tool traffic dropped.
`raw_history`	Every prior turn, role-tagged, tool calls and results included.
`tool_calls`	Prior tool calls and their results, nothing else.
`turn_budget`	Turns used against the cap.
`dispatch_index`	Number of dispatches so far.

A HARNESS STATE

Table 5 and Table 6 summarize the harness state used for our evaluations.

B PROTOCOL FEEDBACK

Table 8 and Table 10 summarize the feedback used for our evaluations.

Table 7: Model prices (USD per 1M tokens) for BrowseComp-Plus.

Model	Role	Input	Cached input	Output
`gpt-5.6-sol`	Planner (main agent)	5.00	0.50	30.00
`gpt-5.6-luna`	Executor (sub-agent)	0.20	0.02	1.20

Table 8: Feedback Signals for Browse-comp Plus.

Signal	Scope	Rule	Shown when	Components
Outcome	Global	Judge verdict and run status: CORRECT (1), INCORRECT (0), or FAILED (0; timeout, error, or no verdict).	Always	All
Extracted answer	Global	Judge-extracted answer, marked as matching or not matching the gold answer; the gold string itself is never shown.	Answer extracted	All
Retrieval metrics	Global	Evidence recall over the qrel evidence set; citation precision and recall.	Incorrect	All
Raw output tail	Global	Last 900 characters of the final output.	No answer extracted	All
Failure class	Global	One of four classes (Table 9), each naming where in the trace to look.	Not correct	All
Task note	Global	Fixed description of the BrowseComp-Plus task.	Not correct	All
Evidence touched	Global	Number of qrel evidence documents touched across the rollout.	Always	Forward, Return
Unused fields	Global	Declared fields never written in any delegation.	Always	Forward, Return
Empty tokens	Global	Template tokens that rendered empty in any delegation.	Always	Forward, Return
Forward chain	Local	Per delegation: goal, search queries issued, and evidence documents newly touched.	Always	Forward
Return chain	Local	Per delegation: report, the next goal, the next queries, and evidence newly touched by the next delegation.	Always	Return

Table 9: Failure classes.

Class	Condition	Where to look
`timeout`	Cancelled at the wall-clock limit	The last search/get_document calls
`retrieval-none`	recall $= 0$	The search queries
`retrieval-partial`	$0 <$ recall < 1	Which searches found evidence and which did not
`retrieval-full`	recall ≥ 1, answer wrong	Reading or reasoning over the opened documents

Table 10: Summary of feedback signals in MAS-PromptBench. All methods give the signals marked π to their prompt components (one system prompt per agent); EvoProtocol additionally gives the signals marked P to its protocol components (one per hand-off), either to *all* of them or only to the one on the hand-off concerned (*own*). Scope: G = global (whole rollout), L = local (one stage or one hand-off). TH = ToolHop, HQ = HotpotQA, GP = GPQA, MA = MATH.

			Dataset				Given to	
Signal	Scope	Content	TH	HQ	GP	MA	π	P
Outcome								
Score	G	Metric score.	✓	✓	✓	✓	✓	all
Answer vs. gold	G	Correctness; predicted and gold answers.	✓	✓	✓	✓	✓	
Other outcome	G/L	Format failures; own-stage output.	✓	✓	✓	✓	✓	
Answer tracking								
Stage answer walk	G	Each stage's answer; where it changed.	✓	✓	✓		✓	
Process signals	G	Option letter per stage; changes; cut-offs.			✓		✓	all
Retrieval verdict	G	Pages reached; answer lost in/after search.		✓			✓	
Answer trail	G	Per hand-off: where the answer is lost.		✓				all
Math hand-offs	G	Per hand-off: answers checked by the scorer.				✓		all
Segment outcome	G	Per segment: errors; answer observed or not.	✓	✓		✓		all
Hand-off answers	L	Sender's and receiver's option letters.			✓			own
Tools								
Tool calls	G/L	Calls, ok/failed (P: all stages; I: own).	✓	✓			✓	own
Repeated failures	L	Tools failing $\geq$2 times; their arguments.	✓	✓			✓	all
Hand-off mechanics								
Interface checks	L	11 gold-free schema, field, template checks.	✓	✓	✓	✓		own

Table 11: Forward and return protocols of the multiple protocol candidate. Every forward variant returns through the single default return protocol.

Variant	Forward fields	Forward harness state	Return fields	Return state
default	task	–	result	–
search_-recovery	task, anchor_-clues, known_-candidates, avoid_queries	original_question, attempted_queries, remaining_budget, repeat_unread_documents, wall_clock_remaining_sec	result	–
evidence_-synthesis	task, candidate_-answer, evidence_-focus, reasoning_-risk	original_question, prior_docids, prior_-findings, remaining_-budget, unread_documents, wall_clock_remaining_sec	result	–
deadline_-rescue	task, best_-candidate, critical_gap	original_question, prior_docids, prior_-findings, remaining_-budget, repeat_unread_-documents, search_-and_get_document_calls, wall_clock_remaining_sec	result	–
chain_-resolution	task, chain_-steps, rarest_-anchor, target_-relation	original_question, prior_docids, prior_-findings, remaining_-budget, wall_clock_-remaining_sec	result	–
surfaced_-document_-triage	task, selection_-rule, critical_fact, search_limit	original_question, attempted_queries, query_yield, remaining_-budget, repeat_unread_-documents, search_-and_get_document_calls, unread_documents, wall_-clock_remaining_sec	result	–

Table 12: communication protocol found by EvoProtocol on MAS-PromptBench. Every found protocol keeps the `task` field; the table lists only the additional fields the sender must write ({`subtask.*`}) and the harness state the message references ({`runtime.*`}). All seed protocol only pass `runtime.prior_stages`.

Task	Edge	Sender-written fields (besides `task`)	Runtime views
ToolHop	planner → caller	`entities_found`, `tool_attempts`	`raw_history`, `turn_budget`
	caller → checker	`entities`, `tool_summary`, `result`	`raw_history`, `tool_calls`, `turn_budget`
	checker → verifier	`key_entities`, `execution_status`	`tool_calls`
HotpotQA	planner → retriever	`plan`	`prior_stages`
	retriever → reasoner	`evidence`	`prior_stages`
	reasoner → writer	`entities`, `facts`, `conclusion`	`prior_stages`
GPQA	analyzer → solver	`analysis`	`raw_history`
	solver → critic	`result`	`prior_stages`
	critic → verifier	`critic_commit`, `critic_concerns`	`prior_stages`
MATH	decomposer → computer	`plan`, `definitions`	`tool_calls`, `prior_stages`
	computer → checker	`result`	`prior_stages`
	checker → verifier	`result`	`prior_stages`
BFCL	analyzer → inspector	`analysis`, `rationale`	—
	inspector → caller	`result`	`raw_history`
	caller → verifier	`plan`	—