MAS-PromptBench: When Does Prompt Optimization Improve Multi-Agent LLM Systems?

Anonymous authors
Paper under double-blind review

Abstract

Multi-agent systems (MAS) offer a scalable path forward for agentic AI, comprising multiple LLM-based agents, each assigned a system prompt and a position within a workflow that governs inter-agent coordination and output aggregation. System prompts thus form a critical and accessible optimization surface: they specify agents' roles and behaviors, enabling system-level improvements without model finetuning. Although prompt optimization has shown substantial potential for single LLMs, extending it to MAS poses distinct challenges, notably an exponentially growing search space. It remains unclear *whether, when, and by how much prompt optimization improves MAS performance, and how sensitive such gains are to system configuration*. In this work, we systematically study system-prompt optimization across a broad range of MAS setups varying in task, workflow, communication protocol, and team size. We benchmark both natural multi-agent extensions of state-of-the-art single-agent prompt optimizers and recent prompt optimizers designed specifically for MAS. The results reveal its potential to unlock significant gains while exposing open challenges, characterizing when and how much prompt optimization helps across diverse MAS settings.

Project Page: https://anonymous.4open.science/w/MAS-PromptBench/

1 Introduction

Agentic AI, as foundation-model-based systems that autonomously plan, use tools, and interact with the real world, is rapidly transforming daily life, industry, and scientific discovery (Lu et al., 2024; Plaat et al., 2025; Anthropic, 2026a). As tasks evolve from human-scale problems to organization-scale challenges that are increasingly complex, open-ended, and time-sensitive, single-agent architectures face fundamental bottlenecks in expertise breadth, context length, and sequential execution (Chen et al., 2024b). In contrast, multi-agent systems (MAS) have emerged as a highly promising paradigm for next-generation agentic AI and artificial superintelligence (ASI) (Anthropic, 2026a; Genewein et al., 2026), offering scalability, timeliness, and reliability through specialization and multimodality, task decomposition and parallelism, and independent cross-checks that strengthen reasoning and factual accuracy (Du et al., 2023). Concretely, a MAS typically comprises multiple LLM-based agents coordinated by a harness that manages communication, task delegation, and output aggregation, with each agent assigned an *instruction set* and a position in a coordination workflow (Anthropic, 2026a). Throughout this paper, we refer to an LLM's instruction set as its *system prompt*, which may be an assembled collection comprising not only the system prompt itself but also other levels of instructions.

Within this MAS design space, *system prompts* provide a critical and accessible optimization surface: they specify each agent's role and behavior (Anthropic, 2026b; OpenAI, 2025; Google, 2026; Meta, 2026), enabling system-level improvement without model fine-tuning. System prompts are among the most accessible levers available to practitioners, who often inherit a fixed configuration and seek improvements without redesigning the underlying architecture; many real-world deployments further preclude topology changes due to safety, compliance, or auditability constraints (Hong et al., 2026). Given the important role of system prompts, automatic prompt optimization has been studied extensively in the single-agent regime, with strong demonstrated benefits (Zhou et al., 2022; Yang et al., 2024; Wang et al., 2024; Khattab et al., 2023; Agrawal et al., 2026).

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.

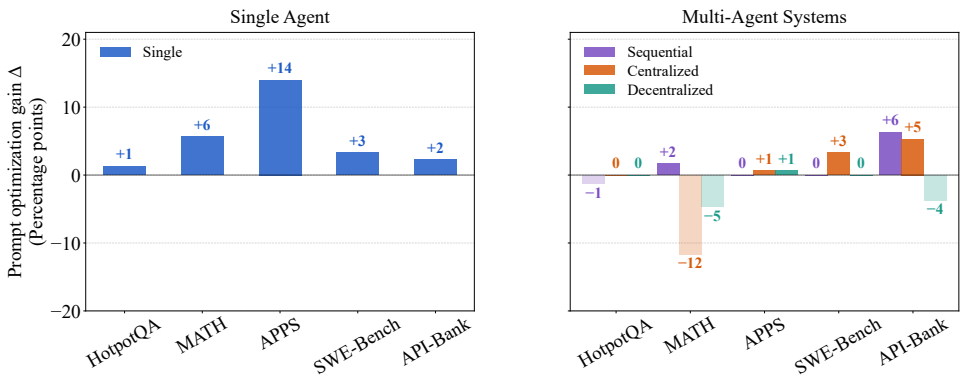


Figure 1: Prompt-optimization gains using a state-of-the-art optimizer MIPRO in single-agent and multi-agent settings. While MIPRO consistently improves single-agent performance across all five diverse tasks, its natural multi-agent extension yields highly variable effects across tasks and workflow topologies, ranging from large gains to severe performance drops.

Whether such gains transfer to the multi-agent setting remains underexplored. Extending prompt optimization to MAS introduces qualitatively new challenges: inter-agent prompt dependencies, compounded by coordination dynamics across multi-turn interactions, induce a combinatorial search space that grows exponentially with the number of agents. As illustrated in Figure 1, the current effect of prompt optimization on MAS varies dramatically across tasks and topologies—ranging from substantial gains to equally severe performance drops. Meanwhile, most influential MAS frameworks—including AutoGen, CrewAI, CAMEL, MetaGPT, ChatDev, and AgentVerse (Wu et al., 2023; CrewAI Inc., 2026; Li et al., 2023a; Hong et al., 2024; Qian et al., 2024; Chen et al., 2024c), as well as collaboration workflows such as debate (Du et al., 2023; Liang et al., 2024)—still rely on manually crafted system prompts. Recent works have begun to address this gap, either by developing dedicated MAS prompt optimization algorithms (Xia et al., 2026; Shen et al., 2025a; Zhang et al., 2026; Hong et al., 2026) or by jointly optimizing prompts alongside orchestration components such as workflow topology (Zhao et al., 2025; Zhou et al., 2026; Hu et al., 2025). Yet these works evaluate on different tasks (e.g., math, coding, and stock trading), configurations, and baselines, making cross-comparison difficult and leaving the current capability boundary of MAS prompt optimization unclear. This leaves a fundamental question open:

How much can prompt optimization help in MAS, and how does its effect vary across configurations?

We address this through a benchmark-driven study that quantifies the gains of eight representative system-prompt optimizers across a diverse set of fixed MAS configurations, thereby characterizing the current capability boundary of MAS prompt optimization. We pinpoint the regimes where prompt optimization offers substantial untapped improvement room and regimes where MAS sensitivity calls for more principled algorithm design. These findings provide a roadmap for designing future prompt optimization algorithms and, more broadly, for MAS design, where prompts are tightly coupled with other configuration choices. Our main contributions are as follows:

- **MAS-PromptBench: a benchmark for MAS prompt optimization.** We introduce a comprehensive benchmark for evaluating system-prompt optimizers for MAS. It spans diverse MAS configurations across three task domains (reasoning, coding, and tool calling), five workflow topologies (comprising both existing and newly constructed systems), communication protocols ranging from free-form to highly structured coordination, and varying team sizes. It also evaluates eight prompt optimizers: two that naturally extend state-of-the-art single-agent methods and six that are designed specifically for MAS. The benchmark provides a foundation for proposing, analyzing, and comparing system-prompt optimization algorithms under controlled MAS configurations.
- **Prompt optimization gains and failures for MAS.** Using this benchmark, we systematically evaluate the performance gains over default system prompts achieved by four representative optimizers: MAS-GEPA and MAS-MIPRO, built on GEPA (Agrawal et al., 2026) and MIPRO (Opsahl-Ong et al., 2024), and the MAS-specific MAPRO (Zhang et al.,

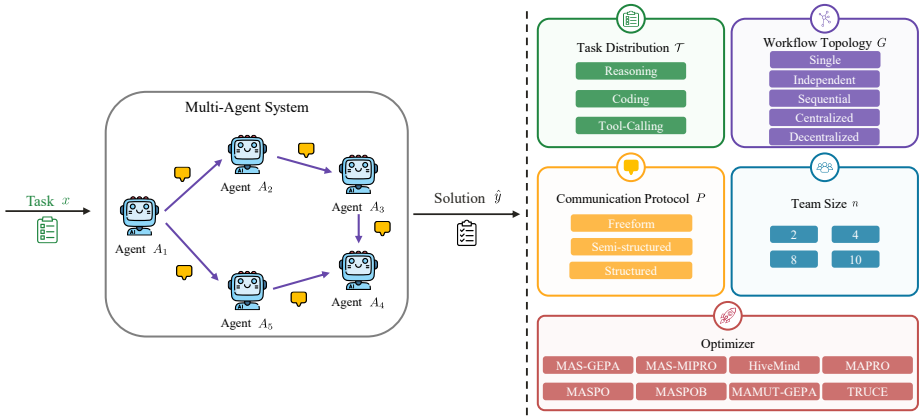


Figure 2: Overview of MAS-PromptBench. Given an input task, a multi-agent system produces a final solution through interactions among LLM-based agents. Using state-of-the-art prompt optimizers, MAS-PromptBench measures prompt-optimization gains across four axes: task distribution, workflow topology, communication protocol, and team size.

2026) and MASPO (Wang et al., 2026). The results highlight the promise of prompt optimization for MAS: improvements reach up to 56.0 percentage points. Yet they also reveal the need for principled algorithms tailored to multi-agent settings, as performance can drop by as much as 12.0 percentage points for certain configurations.

- **Insights into when prompt optimization works for MAS.** Prompt optimization shows greater potential when tasks have explicit, controllable, and verifiable agent-local behaviors, and when communication protocols adopt a free form that leaves agent messages open to revision and gives prompt optimization more room to improve; it also needs to be workflow-topology-aware. In addition, prompt optimization becomes harder as team size grows, confirming the challenges of scaling MAS prompt optimization and motivating principled and more scalable, robust algorithms.

2 Prompt Optimization for Multi-Agent LLM Systems

Prompt optimization for multi-agent systems. We consider an LLM-based MAS (Khattab et al., 2023; Opsahl-Ong et al., 2024; Agrawal et al., 2026), illustrated in Figure 2, represented as the tuple

$$\mathcal{M} = (\mathcal{A},\, G,\, P), \tag{1}$$

where $\mathcal{A} = \{A_1, \ldots, A_n\}$ is an ordered collection of n agents, G denotes the inter-agent coordination workflow, and P denotes the communication protocol between agents. Each agent $A_i = (\theta_i, \pi_i)$ consists of an LLM with model parameters θ_i and a learnable system prompt π_i. We denote $\pi = \{\pi_1, \ldots, \pi_n\}$ as the joint system prompts for all agents. Each task to be solved is drawn as $(x, e) \sim \mathcal{T}$ from a distribution $\mathcal{T}$, where $x \in \mathcal{X}$ is the task input and e is a reference used for evaluation, such as a ground-truth answer or code unit tests. We let $\mathcal{M}(\cdot\,; \pi)$ denote the output function of the MAS $\mathcal{M}$ induced by the joint prompt π, with the model parameters θ held fixed. System-prompt optimization for a MAS is then formulated as the following optimization problem:

$$\max_{\pi}\ \mathbb{E}_{(x,e)\sim\mathcal{T}}[\mu(\mathcal{M}(x;\pi), e)] \quad \text{s.t. } \ell_{\text{rollouts}} \le B, \tag{2}$$

where the metric $\mu : \mathcal{Y} \times \mathcal{E} \to [0, 1]$ evaluates the output $y = \mathcal{M}(x; \pi)$ against the reference e, and ℓ_{rollouts} denotes the number of MAS execution rounds, each comprising one MAS run to solve a task followed by an evaluation of its output.

Remark 1 (*System prompt* as an umbrella term for the instruction set). Within a multi-agent system, each agent is typically prompted with a carefully assembled input package determined by the harness, which may include the user's request, a layered set of instructions, task-specific skills, relevant memory, retrieved context, and available tools. We focus on the instruction set, which typically comprises four components that differ in authority, scope, and purpose, forming a hierarchy that

guides behavior from global constraints to task-specific procedures. At the top, system prompts define universal rules, such as safety and tool-use protocols; developer or product instructions then specify application-level operation, including the agent's role and workflow; project instructions specify task-level conventions, such as coding standards and testing procedures; and finally, skills provide reusable guidance for particular task classes, such as data analysis or code modification. Throughout this paper, we use *system prompt* as an umbrella term for the instruction set excluding skills, without distinguishing among them. In practice, agentic systems may employ multiple instruction layers; our formulation extends naturally to optimizing any subset of them, which we leave to future work.

Metric: prompt optimization gains in MAS. In this work, we conduct a systematic study of whether, when, and to what extent system-prompt optimization improves MAS performance across different MAS configurations. To this end, we investigate diverse MAS configurations determined by four critical components $(\mathcal{T}, G, n, P)$: the task distribution $\mathcal{T}$, the coordination workflow or topology G, the team size n, and the communication protocol P. Any prompt optimizer can, in principle, be used to optimize prompts by solving equation 2. In this work, we primarily focus on four prompt optimizers: MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO (Section 3). Let π^0 denote the initialized (unoptimized) system prompt and $\pi^\star$ the optimized prompt obtained by solving equation 2. Fixing an optimizer for solving equation 2 and the base model, the *prompt-optimization gain* for the MAS with a configuration $(\mathcal{T}, G, n, P)$ is then defined as

$$\Delta(\mathcal{T}, G, n, P) \;:=\; \mathbb{E}_{(x,e)\sim\mathcal{T}}\big[\mu(\mathcal{M}(x;\pi^\star), e) \;-\; \mu\big(\mathcal{M}(x;\pi^0), e\big)\big]. \tag{3}$$

3 MAS-PROMPTBENCH: PROMPT OPTIMIZATION FOR MAS BENCHMARK

While benchmarks and evaluation protocols exist for single-agent prompt optimization, comparable resources for multi-agent systems remain underdeveloped. We fill this gap by introducing a benchmark designed to support extensive and controlled investigation for prompt optimization across diverse MAS configurations, summarized in Table 1 and Figure 2.

Table 1: Overview of the modular configuration of MAS-PromptBench.

Factor	**#**	**Details**
MAS Framework	4	LangGraph (LangChain Inc., 2026), CrewAI (CrewAI Inc., 2026), AutoGen (Wu et al., 2023), OpenAI Agents SDK (OpenAI, 2026)
Task	9	*Reasoning* (3): GPQA-Diamond (Rein et al., 2023), HotpotQA (Yang et al., 2018), MATH (Hendrycks et al., 2021b); *Coding* (3): LiveCodeBench (Jain et al., 2025), APPS (Hendrycks et al., 2021a), SWE-Bench Verified (Jimenez et al., 2024); *Tool-calling* (3): BFCL (Patil et al., 2025), ToolHop (Ye et al., 2025), API-Bank (Li et al., 2023b)
Topology	5	Single, Independent, Sequential, Centralized, Decentralized
Communication	3	Freeform, Semi-structured, Structured
Team size	4	$n \in \{2, 4, 8, 10\}$
Optimizer	8	*Extensions of single-agent optimizers* (2): MAS-GEPA (Agrawal et al., 2026), MAS-MIPRO (Opsahl-Ong et al., 2024); *multi-agent optimizers* (6): HiveMind (Xia et al., 2026), MAPRO (Zhang et al., 2026), MASPO (Wang et al., 2026), MASPOB (Hong et al., 2026), MAMUT-GEPA (Herrera et al., 2026), TRUCE (Wu et al., 2026)

The proposed benchmark evaluates the prompt-optimization gain using the metric in equation 3, which compares the optimized prompts $\pi^\star$ against the initialized prompts π^0 for a fixed prompt optimizer and MAS configuration $(\mathcal{T}, G, n, P)$. This controlled protocol makes the benchmark useful in at least two ways. First, given any system-prompt optimizer, it provides an extensive testbed across diverse MAS configurations, enabling direct evaluation of how optimization gains vary with task, topology, communication structure, and team size. Second, its modular design supports controlled, component-level studies: one can vary a single MAS factor—such as the optimizer, topology, protocol, or team size—while holding all others fixed to isolate its effect on system-level performance. The benchmark is also flexible and extensible across all of these components, as new tasks or MAS configurations can be introduced as additional configuration values, allowing

Table 2: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO for nine datasets on existing MAS frameworks. Each cell reports baseline / optimized performance, followed by the signed change Δ in percentage points, with the standard deviation of Δ across three optimizer seeds as a subscript. Blue indicates improvement, orange indicates regression, and gray indicates no change. Average Max: mean best-optimizer gain over topologies; darker blue marks larger gains. Dashes: not applicable (MAPRO and MASPO require multiple agents).

Topology	Method	*Reasoning*			*Coding*			*Tool-Calling*		
		GPQA-Diamond (Acc.)	HotpotQA (EM)	MATH (Acc.)	LiveCodeBench (pass@1)	APPS (pass@1)	SWE-Bench Verified (Resolved)	BFCL (Acc.)	ToolHop (Acc.)	API-Bank (Acc.)
Single (LangGraph)	MAS-GEPA	76.3 / 72.3 $-4.0_{\pm 3.5}$	55.0 / 56.3 $+1.3_{\pm 2.3}$	67.7 / 71.7 $+4.0_{\pm 3.5}$	68.0 / 84.0 $+16.0_{\pm 6.0}$	60.0 / 60.0 $0.0_{\pm 0.0}$	40.0 / 40.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	66.0 / 65.0 $-1.0_{\pm 1.7}$	78.7 / 76.3 $-2.3_{\pm 4.0}$
	MAS-MIPRO	78.7 / 78.7 $0.0_{\pm 0.0}$	54.7 / 56.0 $+1.3_{\pm 1.2}$	67.7 / 73.3 $+5.7_{\pm 4.5}$	84.0 / 84.0 $0.0_{\pm 0.0}$	64.0 / 78.0 $+14.0_{\pm 2.0}$	42.2 / 45.6 $+3.3_{\pm 3.3}$	54.0 / 54.0 $0.0_{\pm 0.0}$	68.0 / 68.0 $0.0_{\pm 0.0}$	79.3 / 81.7 $+2.3_{\pm 2.9}$
	MAPRO	–	–	–	–	–	–	–	–	–
	MASPO	–	–	–	–	–	–	–	–	–
Independent (LangGraph)	MAS-GEPA	77.3 / 77.3 $0.0_{\pm 2.6}$	39.3 / 50.7 $+11.3_{\pm 3.8}$	55.7 / 61.7 $+6.0_{\pm 7.2}$	86.0 / 86.0 $0.0_{\pm 0.0}$	74.0 / 74.0 $0.0_{\pm 0.0}$	56.7 / 56.7 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	63.7 / 64.0 $+0.3_{\pm 0.6}$	76.0 / 81.0 $+5.0_{\pm 1.7}$
	MAS-MIPRO	78.0 / 76.7 $-1.3_{\pm 4.6}$	53.1 / 53.1 $0.0_{\pm 0.0}$	74.3 / 75.7 $+1.3_{\pm 2.3}$	86.0 / 86.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$	46.7 / 56.7 $+10.0_{\pm 3.3}$	54.0 / 54.0 $0.0_{\pm 0.0}$	64.7 / 64.7 $0.0_{\pm 0.0}$	76.0 / 76.7 $+0.7_{\pm 1.2}$
	MAPRO	78.3 / 78.0 $-0.3_{\pm 0.6}$	46.7 / 46.7 $0.0_{\pm 0.0}$	45.3 / 47.3 $+2.0_{\pm 3.5}$	90.0 / 90.0 $0.0_{\pm 0.0}$	74.0 / 74.0 $0.0_{\pm 0.0}$	60.0 / 56.7 $-3.3_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	64.7 / 65.3 $+0.7_{\pm 1.2}$	76.3 / 76.3 $0.0_{\pm 0.0}$
	MASPO	78.3 / 78.7 $+0.3_{\pm 0.6}$	44.2 / 49.2 $+5.0_{\pm 4.6}$	45.0 / 45.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	78.0 / 76.0 $-2.0_{\pm 2.0}$	46.7 / 50.0 $+3.3_{\pm 3.3}$	54.0 / 54.3 $+0.3_{\pm 0.6}$	64.0 / 65.7 $+1.7_{\pm 1.5}$	76.0 / 77.7 $+1.7_{\pm 2.9}$
Sequential (CrewAI)	MAS-GEPA	71.7 / 70.0 $-1.7_{\pm 2.9}$	49.3 / 49.3 $0.0_{\pm 0.0}$	65.7 / 70.0 $+4.3_{\pm 5.7}$	88.0 / 88.0 $0.0_{\pm 0.0}$	70.0 / 64.0 $-6.0_{\pm 2.0}$	46.7 / 60.0 $+13.3_{\pm 13.3}$	78.7 / 82.3 $+3.7_{\pm 3.5}$	70.7 / 71.3 $+0.7_{\pm 3.1}$	67.3 / 66.7 $-0.7_{\pm 2.9}$
	MAS-MIPRO	76.3 / 76.7 $+0.3_{\pm 3.2}$	49.3 / 48.0 $-1.3_{\pm 2.3}$	65.0 / 66.7 $+1.7_{\pm 1.5}$	86.0 / 86.0 $0.0_{\pm 0.0}$	64.0 / 64.0 $0.0_{\pm 0.0}$	56.7 / 56.7 $0.0_{\pm 0.0}$	79.3 / 86.0 $+6.7_{\pm 1.5}$	70.0 / 70.0 $0.0_{\pm 1.0}$	66.7 / 73.0 $+6.3_{\pm 5.7}$
	MAPRO	74.7 / 71.7 $-3.0_{\pm 7.0}$	49.3 / 49.3 $0.0_{\pm 0.0}$	65.0 / 66.3 $+1.3_{\pm 2.3}$	88.0 / 87.3 $-0.7_{\pm 6.1}$	67.3 / 67.3 $0.0_{\pm 0.0}$	56.7 / 56.7 $0.0_{\pm 0.0}$	80.3 / 82.7 $+2.3_{\pm 2.1}$	70.7 / 70.7 $0.0_{\pm 0.0}$	67.0 / 68.3 $+1.3_{\pm 2.3}$
	MASPO	77.3 / 74.3 $-3.0_{\pm 2.6}$	49.0 / 49.0 $0.0_{\pm 0.0}$	65.0 / 62.0 $-3.0_{\pm 4.4}$	87.3 / 88.7 $+1.3_{\pm 6.4}$	64.0 / 64.0 $0.0_{\pm 0.0}$	63.3 / 56.7 $-6.7_{\pm 3.3}$	81.3 / 81.7 $+0.3_{\pm 0.6}$	70.3 / 70.3 $0.0_{\pm 0.0}$	66.3 / 66.3 $0.0_{\pm 0.0}$
Centralized (AutoGen)	MAS-GEPA	77.0 / 76.5 $-0.5_{\pm 0.7}$	44.0 / 44.0 $0.0_{\pm 0.0}$	44.7 / 49.7 $+5.0_{\pm 8.7}$	88.0 / 88.0 $0.0_{\pm 0.0}$	71.4 / 71.4 $0.0_{\pm 0.0}$	30.0 / 30.0 $0.0_{\pm 0.0}$	79.7 / 83.3 $+3.7_{\pm 8.7}$	70.3 / 71.3 $+1.0_{\pm 1.7}$	74.3 / 75.3 $+1.0_{\pm 1.7}$
	MAS-MIPRO	76.3 / 76.3 $0.0_{\pm 0.0}$	51.2 / 51.2 $0.0_{\pm 0.0}$	65.0 / 53.3 $-11.7_{\pm 10.4}$	86.0 / 86.0 $0.0_{\pm 0.0}$	67.3 / 68.0 $+0.7_{\pm 3.1}$	23.3 / 26.7 $+3.3_{\pm 3.3}$	77.0 / 84.3 $+7.3_{\pm 6.4}$	69.0 / 70.3 $+1.3_{\pm 2.3}$	71.7 / 77.0 $+5.3_{\pm 4.7}$
	MAPRO	76.7 / 76.7 $0.0_{\pm 0.0}$	58.6 / 58.6 $0.0_{\pm 0.0}$	56.7 / 59.7 $+3.0_{\pm 3.6}$	88.0 / 88.0 $0.0_{\pm 0.0}$	71.4 / 73.5 $+2.0_{\pm 2.0}$	26.7 / 40.0 $+13.3_{\pm 3.3}$	78.3 / 78.0 $-0.3_{\pm 0.6}$	70.3 / 70.0 $-0.3_{\pm 0.6}$	73.0 / 72.7 $-0.3_{\pm 0.6}$
	MASPO	78.5 / 73.5 $-5.0_{\pm 7.1}$	47.8 / 47.1 $-0.7_{\pm 1.2}$	63.9 / 64.3 $+0.3_{\pm 0.6}$	86.0 / 88.0 $+2.0_{\pm 2.0}$	64.0 / 64.0 $0.0_{\pm 0.0}$	23.3 / 43.3 $+20.0_{\pm 3.3}$	78.3 / 87.7 $+9.3_{\pm 2.3}$	69.7 / 70.7 $+1.0_{\pm 1.7}$	73.3 / 74.3 $+1.0_{\pm 1.7}$
Decentralized (OpenAI SDK)	MAS-GEPA	67.7 / 67.7 $0.0_{\pm 0.0}$	41.0 / 44.3 $+3.3_{\pm 2.1}$	59.7 / 60.0 $+0.3_{\pm 0.6}$	40.0 / 40.0 $0.0_{\pm 0.0}$	40.8 / 40.8 $0.0_{\pm 0.0}$	30.0 / 32.2 $+2.2_{\pm 1.9}$	73.3 / 73.3 $0.0_{\pm 0.0}$	42.5 / 42.5 $0.0_{\pm 0.0}$	25.3 / 33.7 $+8.3_{\pm 2.1}$
	MAS-MIPRO	67.7 / 67.7 $0.0_{\pm 0.0}$	41.0 / 41.0 $0.0_{\pm 0.0}$	59.7 / 55.0 $-4.7_{\pm 8.1}$	50.7 / 50.0 $-0.7_{\pm 3.1}$	40.7 / 41.3 $+0.7_{\pm 5.0}$	30.0 / 30.0 $0.0_{\pm 0.0}$	74.3 / 62.3 $-12.0_{\pm 18.2}$	44.3 / 43.9 $-0.3_{\pm 0.6}$	25.3 / 21.7 $-3.7_{\pm 6.4}$
	MAPRO	67.3 / 67.7 $+0.3_{\pm 0.6}$	41.0 / 41.0 $0.0_{\pm 0.0}$	52.0 / 52.7 $+0.7_{\pm 1.2}$	62.0 / 62.0 $0.0_{\pm 0.0}$	41.3 / 42.0 $+0.7_{\pm 6.4}$	30.0 / 35.6 $+5.6_{\pm 1.9}$	73.3 / 73.3 $0.0_{\pm 0.0}$	42.5 / 42.5 $0.0_{\pm 0.0}$	25.3 / 36.0 $+10.7_{\pm 10.3}$
	MASPO	67.7 / 68.3 $+0.7_{\pm 1.2}$	41.0 / 41.0 $0.0_{\pm 0.0}$	52.7 / 52.7 $0.0_{\pm 0.0}$	51.3 / 50.7 $-0.7_{\pm 1.2}$	37.3 / 36.0 $-1.3_{\pm 3.1}$	30.0 / 34.4 $+4.4_{\pm 1.9}$	73.3 / 74.0 $+0.7_{\pm 1.2}$	42.5 / 63.0 $+20.5_{\pm 0.7}$	25.3 / 47.7 $+22.3_{\pm 5.1}$
Average Max		+0.3	+3.2	+4.3	+3.9	+3.3	+10.4	+3.4	+4.8	+8.2

it to be tailored to different application domains and user requirements. We provide eight prompt optimizers: MAS-GEPA and MAS-MIPRO are multi-agent extensions of the state-of-the-art single-agent prompt optimizers GEPA and MIPRO, respectively; the other six are designed specifically for MAS (HiveMind, MAPRO, MASPO, MASPOB, MAMUT-GEPA, and TRUCE). Our main experiments use MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO. This benchmark evaluates each task dataset using its official evaluation protocol; detailed metric definitions and implementations are provided in Appendix C.2.

4 Empirical Study of Prompt Optimization in MAS

Armed with the MAS-PromptBench benchmark, in this section we conduct a systematic study to answer: *How much can prompt optimization help in MAS, and how does its effect vary across configurations?* We subsequently investigate the four critical MAS configuration factors: task (Sec. 4.1), workflow topology (Sec. 4.2), communication protocol (Sec. 4.3), and team size (Sec. 4.4). Sec. 4.1–Sec. 4.4 use four prompt optimizers: MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO. Sec. 4.5 further ablates the prompt optimizer and the task model. Full results for Sec. 4.2–Sec. 4.5 are reported in Appendix B. MAS-GEPA and MAS-MIPRO optimize each agent's system prompt as a separate module, using feedback from MAS execution and the agent's own traces. Details are in Appendix C.4. MAPRO and MASPO instead optimize the agents' system prompts jointly.

4.1 Task

We first study how prompt-optimization gains vary across nine tasks spanning three domains: reasoning, coding, and tool-calling. In this subsection on tasks, we evaluate a range of popular existing MAS frameworks with naturally differing topologies, shown in Table 2; in all remaining studies on MAS configurations, we instead use the LangGraph framework to construct the different configurations, for flexibility and fairness. Table 2 shows that system-prompt optimization is broadly promising across diverse tasks: averaged over workflows, the best optimizer per workflow improves performance on all nine tasks, with the largest average gain of $+10.4$ points on SWE-Bench. Individual MAS configurations show even larger gains: with MASPO, the Decentralized workflow improves by $+22.3$ on API-Bank and $+20.5$ on ToolHop, and the Centralized workflow by $+20.0$ on SWE-Bench.

The gains are larger for coding and tool-calling tasks than for reasoning tasks. At the task level, averaging the gain of the best optimizer over topologies (Average Max in Table 2), the largest gains for coding and tool-calling are $+10.4$ points on SWE-Bench and $+8.2$ points on API-Bank, respectively, whereas the largest among reasoning tasks is only $+4.3$ points on MATH. At the topology-configuration level, coding and tool-calling tasks achieve maximum gains of $+20.0$ points

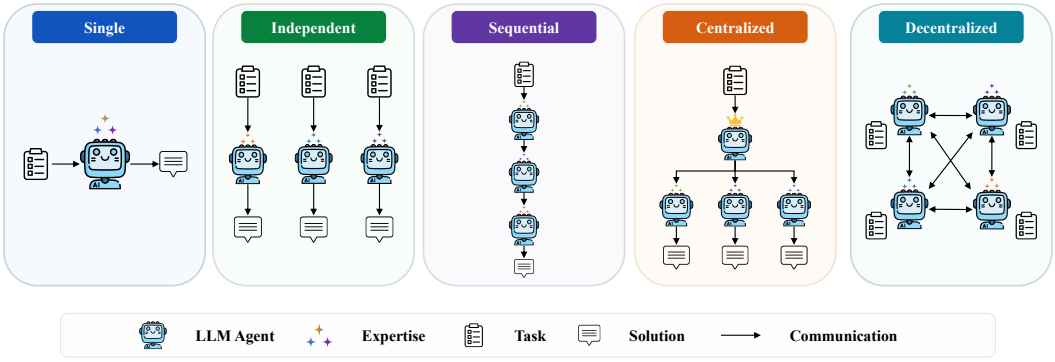


Figure 3: The five workflow topologies in MAS-PromptBench. Arrows indicate message flow.

on Centralized SWE-Bench and $+22.3$ points on Decentralized API-Bank, both with MASPO, whereas reasoning tasks reach only $+11.3$ points on Independent HotpotQA with MAS-GEPA. The same trend holds at the domain level: averaged over their tasks, the Average Max gains of coding and tool-calling benchmarks are $+5.9$ and $+5.5$ points, respectively, compared with only $+2.6$ points for reasoning benchmarks.

We hypothesize that the difference arises from the extent to which each task can be decomposed into an explicit routine with controllable local behaviors. Coding tasks expose verifiable artifacts—small program steps can be checked by compilation and tests, while tool-calling tasks offer structured and clear interfaces through explicit function names and outcome formats that system prompts can directly shape. Such behaviors propagate through the MAS workflow with little ambiguity, allowing local prompt improvements to survive downstream coordination. Reasoning tasks, in contrast, rely on correlated logical steps with implicit intermediate feedback, so local improvements or errors are often discarded, overwritten, or amplified before reaching the final answer. Prompt optimization is thus most effective when tasks provide structured interfaces through which agent-level changes can be clearly controlled, preserved, and transferred across agents.

> ***Takeaway.*** Prompt optimization shows greater potential on tasks with explicit, controllable, and verifiable agent-local behaviors, such as coding and tool-calling, than on reasoning tasks.

4.2 Workflow Topology

A workflow topology refers to the inter-agent coordination graph G, which determines how agent outputs (*messages*) are routed, combined, and exposed to other agents en route to the final outcome.

To study the room for improvement and the level of difficulty across diverse MAS topologies, we evaluate prompt-optimization gains under the following four multi-agent topologies along with a single-agent baseline, as illustrated in Figure 3.

- *Single:* A single LLM serves as the baseline.
- *Independent:* n agents solve the task in parallel without inter-agent communication, and their outputs are aggregated by majority vote.
- *Sequential:* Agents form a directed chain $A_1 \to A_2 \to \cdots \to A_n$ with no backward edges; each agent receives the previous agent's output as input toward the final answer.
- *Centralized:* A coordinator dispatches subtasks to sub-agents $A_1, \ldots, A_n$, collects their outputs, and aggregates them into the final answer; sub-agents do not communicate with one another throughout the process.
- *Decentralized:* All n agents communicate over a fully connected graph and exchange messages for a fixed number of rounds, after which their final-round outputs are aggregated by majority vote for question-answering tasks or best-of-N test-pass for coding tasks.

As shown in Figure 4, the gain of each prompt optimizer varies considerably across MAS topologies. MAS-GEPA gains $+6.0$ points on Sequential but only $+0.2$ on Centralized, and MAS-MIPRO gains

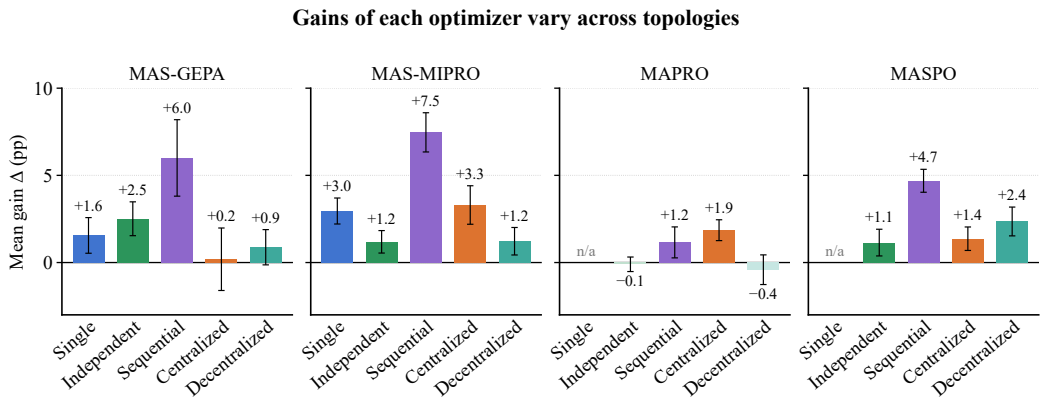


Figure 4: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO across workflow topologies, averaged over nine tasks. Gains vary considerably across topologies, suggesting that MAS prompt optimization needs topology-aware optimizers.

$+7.5$ points on Sequential but only $+1.2$ on Independent and Decentralized. The best optimizer also changes with the topology: MAS-GEPA leads on Independent, MAS-MIPRO on Sequential and Centralized, and MASPO on Decentralized. MAPRO and MASPO, although designed for MAS, still range from -0.4 to $+1.9$ points and from $+1.1$ to $+4.7$ points across topologies, respectively. No evaluated optimizer transfers uniformly across topologies, which motivates topology-aware prompt optimizers rather than a one-size-fits-all method.

In particular, optimization on the Sequential topology yields the largest gains—reaching $+7.5$ points for MAS-MIPRO and $+4.8$ points on average across the four optimizers—suggesting that an improved prompt early in the chain propagates to every downstream agent. The Independent and Decentralized topologies, in contrast, yield the smallest gains, averaging only $+1.2$ and $+1.0$ points, with MAPRO at -0.1 and -0.4 points, respectively, suggesting that aggregating parallel agent outputs may absorb much of the effect of prompt revisions.

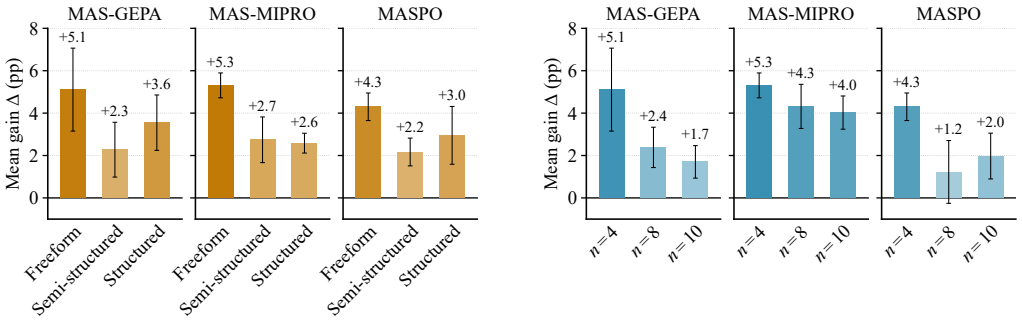
> **Takeaway.** Multi-agent systems need topology-aware prompt optimizers.

4.3 Communication Protocol

A communication protocol specifies the format of inter-agent messages. Since downstream agents observe only the information explicitly written by upstream agents, an underspecified or overly redundant protocol may obscure salient information or direct attention to irrelevant details. To study how communication structure affects prompt optimization, we consider three protocols with increasing levels of structure; concrete examples of each protocol are provided in Appendix C.3:

- *Freeform:* Agents exchange unrestricted natural-language messages with no required fields or templates. This protocol gives agents maximum flexibility, but downstream agents must infer which information is most relevant.
- *Semi-structured:* Agents communicate through a small set of prescribed slots that summarize the sender's status, evidence, confidence, and intended next step. Each slot is still filled in natural language, making the message easier to scan while preserving flexibility for task-specific details.
- *Structured:* Agents communicate using a JSON-style format with a fixed set of predefined slots for critical information, such as status, summary, confidence level, supporting evidence, and next action. Unlike the semi-structured protocol, each slot's value follows a more constrained format drawn from a predefined, finite set of options. This makes message organization more consistent and reduces ambiguity across agents, but also limits how freely agents can express task-specific details.

As shown in Figure 5a, freeform communication protocols yield consistently larger prompt-optimization gains. Averaged over MAS-GEPA, MAS-MIPRO, and MASPO, the gain is $+4.9$



(a) Communication protocols (b) Team sizes

Figure 5: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, and MASPO, averaged over HotpotQA, LiveCodeBench, and BFCL and four MAS topologies. (a) Across communication protocols, Freeform yields the largest gain for every optimizer. (b) Across team sizes, every optimizer gains less with ten agents than with four. Darker bars mark larger gains.

points under Freeform messages, compared with $+2.4$ under Semi-structured messages and $+3.0$ under Structured messages. The advantage is largest for MAS-MIPRO, where the gain under Freeform messages ($+5.3$) is about twice that under Semi-structured ($+2.7$) and Structured ($+2.6$) messages. Here, freeform messages let the optimizer revise both the content and the form of what each agent passes downstream, so that a revised prompt directly changes what the next agent receives. In contrast, gains under Semi-structured and Structured messages differ by at most 1.3 points for each optimizer, where a fixed template already determines much of each message, leaving less for prompt optimization to change. Overall, prompt optimization is most effective when communication protocols leave message content open to revision, allowing local prompt improvements to reshape what agents hand to one another as they propagate through the MAS workflow.

Takeaway. Freeform communication leaves agent messages open to revision, giving MAS prompt optimization more room to improve than protocols that fix message structure in advance.

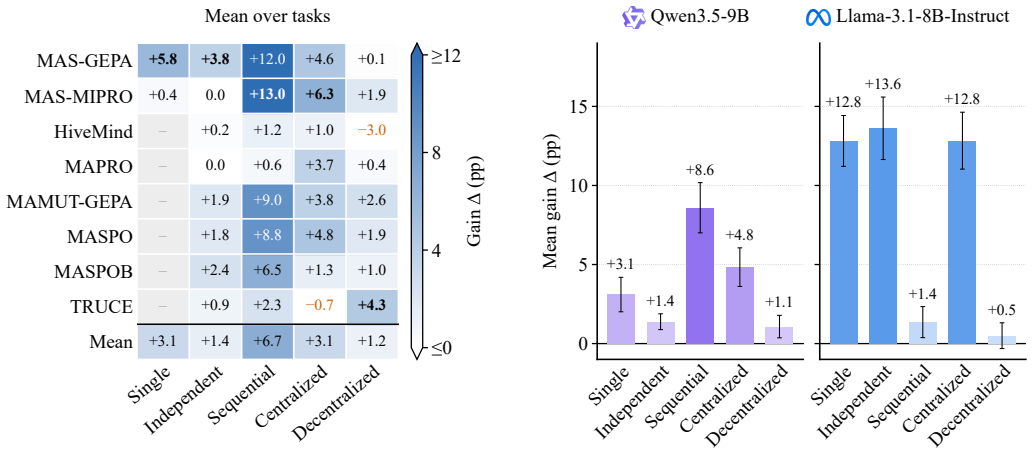
4.4 Team Size

In this section, we study whether prompt-optimization gains increase with team size due to improved scalability, or decrease as coordination overhead grows. We vary the number of agents $n \in \{4, 8, 10\}$. Figure 5b shows that as team size increases, prompt-optimization gains generally decrease, indicating that it becomes more challenging for prompt optimization to translate into system-level gains, as agent-local improvements may be diluted or lost through increased coordination complexity. Averaged over MAS-GEPA, MAS-MIPRO, and MASPO, gains fall from $+4.9$ points at n=4 to $+2.6$ at n=8 and remain at $+2.6$ at n=10. This pattern suggests that adding more agents does not necessarily create more opportunities for prompt optimization, at least for current optimizers. While larger teams may enable finer-grained specialization, they also introduce more handoffs and intermediate states, making local improvements harder to preserve throughout the workflow. This effect is especially clear for MAS-GEPA, where gains fall from $+5.1$ at four agents to $+2.4$ at eight and $+1.7$ at ten. In contrast, MAS-MIPRO retains most of its gain across all team sizes ($+5.3$, $+4.3$, and $+4.0$), indicating that the effect of team size also depends on the prompt optimizer.

Takeaway. Larger team size increases the challenge of prompt optimization for MAS, as local agent improvements may fail to produce system-level gains.

4.5 Ablation Study

Ablation of prompt optimizers. The previous subsections focus on four representative prompt optimizers: MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO. To assess whether these findings are specific to these optimizers or generalize to other prompt optimizers for MAS, we additionally evaluate HiveMind, MAMUT-GEPA, MASPOB, and TRUCE. We conduct the same workflow-topology



(a) Prompt-optimizer ablation (b) Task-model ablation

Figure 6: (a) Prompt-optimization gains of eight prompt optimizers across workflow topologies, averaged over HotpotQA, LiveCodeBench, and BFCL. Bold marks the best optimizer per topology, Mean averages over optimizers, and dashes mark inapplicable settings. (b) Prompt-optimization gains with Qwen3.5-9B and Llama-3.1-8B-Instruct as the task model, averaged over HotpotQA, LiveCodeBench, and BFCL and over MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO (only MAS-GEPA and MAS-MIPRO for Single). Darker bars mark larger gains.

experiments as described in Sec. 4.2 on HotpotQA, LiveCodeBench, and BFCL, with results presented in Figure 6a. The results exhibit trends highly consistent with those observed for the four main optimizers. The Sequential topology yields the largest gain for six of the eight optimizers and $+6.7$ points on average, whereas the Independent and Decentralized topologies yield the smallest average gains at $+1.4$ and $+1.2$ points, respectively. As with the four main optimizers, the best optimizer also changes with the topology: MAS-GEPA leads on Single and Independent, MAS-MIPRO on Sequential and Centralized, and TRUCE on Decentralized. This ablation study suggests that our findings are not tied to a specific prompt optimizer. Instead, they likely reflect inherent challenges in prompt optimization for MAS and may provide useful guidance for the design of future topology-aware optimizers.

Ablation of task model. All experiments in the previous subsections run every agent on Qwen3.5-9B. To examine whether the observed effects depend on this task model, we replace it with Llama-3.1-8B-Instruct while keeping the prompt optimizers, tasks, MAS configurations, and reflection model unchanged. We repeat the workflow-topology experiments of Sec. 4.2 on HotpotQA, LiveCodeBench, and BFCL with MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO, and report the results in Figure 6b. For both task models, prompt-optimization gains vary considerably across topologies, but the topologies rank differently. Sequential is the most improved topology for Qwen3.5-9B at $+8.6$ points, yet gains only $+1.4$ points for Llama-3.1-8B-Instruct, which instead improves most on Independent and Centralized, at $+13.6$ and $+12.8$ points, respectively. Decentralized, by comparison, remains the least improved topology for both models, at $+1.1$ and $+0.5$ points. Hence, the dependence of gains on topology persists across task models, whereas the topology that benefits most varies with the task model, suggesting that topology-aware prompt optimizers should also account for the task model.

5 Conclusion

For multi-agent LLM systems (MAS), we study system prompts as a critical and accessible optimization surface that requires no fine-tuning. We build MAS-PromptBench, a benchmark for MAS prompt optimization spanning diverse tasks, workflow topologies, communication protocols, team sizes, and optimizers. Prompt optimization improves MAS performance by up to 56.0 points, yet gains vary widely across topologies and performance can drop by as much as 12.0 points, calling for principled algorithms tailored to MAS. It is most effective on tasks with explicit, controllable, and verifiable agent-level behaviors and under communication protocols that leave message content open to revision, whereas larger teams add coordination overhead that hinders optimization.

AI USE STATEMENT

We used generative AI tools only to proofread and edit the paper text, to format figures and LaTeX tables, and to debug our implementation code. We did not use them to develop ideas, design experiments, write the method implementation, or generate or process data; all research decisions and conclusions are our own. Large language models are also the object of study (Appendices C.5 and D.1) rather than research assistance. We reviewed all AI-assisted edits and code fixes and take responsibility for the final content of this work.

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.

Anthropic. Claude code, 2026a. URL https://github.com/anthropics/claude-code. GitHub repository. Accessed: 2026-06-17.

Anthropic. System prompts. https://platform.claude.com/docs/en/release-notes/system-prompts, 2026b. Accessed: 2026-05-16.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *Advances in Neural Information Processing Systems*, 38, 2026.

Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Xiaoqian Liu, Tong Xiao, and Jingbo Zhu. Efficient prompting methods for large language models: A survey. *arXiv preprint arXiv:2404.01077*, 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.

Shuaihang Chen, Yuanxing Liu, Wei Han, Weinan Zhang, and Ting Liu. A survey on llm-based multi-agent system: Recent advances and new frontiers in application. *arXiv preprint arXiv:2412.17481*, 2024b.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*, volume 2024, pp. 20094–20136, 2024c.

CrewAI Inc. Crewai: Framework for orchestrating role-playing autonomous ai agents. https://github.com/crewaiinc/crewai, 2026. Accessed: 2026-04-30.

Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 conference on empirical methods in natural language processing*, pp. 3369–3391, 2022.

Victor Dibia, Jingya Chen, Gagan Bansal, Suff Syed, Adam Fourney, Erkang Zhu, Chi Wang, and Saleema Amershi. Autogen studio: A no-code developer tool for building and debugging multi-agent systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 72–79, 2024.

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

Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fourney, Jack Gerrits, Erkang Zhu, and Saleema Amershi. Interactive debugging and steering of multi-agent ai systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pp. 1–15, 2025.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Tim Genewein, Matija Franklin, Alexander Lerchner, Laurent Orseau, Samuel Albanie, Adam Bales, Cole Wyeth, Stephanie Chan, Iason Gabriel, Joel Z Leibo, et al. From agi to asi. *arXiv preprint arXiv:2606.12683*, 2026.

Google. Gemini generatecontent api. https://ai.google.dev/gemini-api/docs/text-generation, 2026. Accessed: 2026-05-16.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 34133–34156, 2024.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021a.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021b.

Alejandro Breen Herrera, Aayush Sheth, Steven G Xu, Zhucheng Zhan, Charles Wright, Marcus Yearwood, Hongtai Wei, Sudeep Das, Danny Nightingale, Meg Watson, et al. Build, judge, optimize: A blueprint for continuous improvement of multi-agent consumer assistants. *arXiv preprint arXiv:2603.03565*, 2026.

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

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

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

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

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.

Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, et al. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*, 2025.

LangChain Inc. Langgraph: Build resilient language agents as stateful graph workflows. https://github.com/langchain-ai/langgraph, 2026. Accessed: 2026-04-30.

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

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 3102–3116, 2023b.

Wenwu Li, Yuran Song, Mingze Zhao, Bo Jin, and Wenhao Li. Unifying temporal and structural credit assignment in llm-based multi-agent prompt optimization. *arXiv preprint arXiv:2605.30227*, 2026.

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.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Meta. Llama 4: Model cards and prompt formats. https://www.llama.com/docs/model-cards-and-prompt-formats/llama4/, 2026. Accessed: 2026-05-16.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations*, volume 2024, pp. 9025–9049, 2024.

OpenAI. Model spec. https://model-spec.openai.com/2025-12-18.html, December 2025. Accessed: 2026-05-16.

OpenAI. The next evolution of the agents sdk. https://openai.com/index/the-next-evolution-of-the-agents-sdk/, 2026. Accessed: 2026-04-30.

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.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Aske Plaat, Max Van Duijn, Niki Van Stein, Mike Preuss, Peter Van der Putten, and Kees Joost Batenburg. Agentic large language models, a survey. *Journal of Artificial Intelligence Research*, 84, 2025.

Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. Grips: Gradient-free, edit-based instruction search for prompting large language models. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 3845–3864, 2023.

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.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 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, et al. Scaling large language model-based multi-agent collaboration. In *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025.

Kiran Ramnath, Kang Zhou, Sheng Guan, Soumya Smruti Mishra, Xuan Qi, Zhengyuan Shen, Shuai Wang, Sangmin Woo, Sullam Jeoung, Yawei Wang, et al. A systematic survey of automatic prompt optimization techniques. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33066–33098, 2025.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Ming Shen, Raphael Shu, Anurag Pratik, James Gung, Yubin Ge, Monica Sunkara, and Yi Zhang. Optimizing llm-based multi-agent system with textual feedback: A case study on software development. *arXiv preprint arXiv:2505.16086*, 2025a.

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

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Y Zou. Mixture-of-agents enhances large language model capabilities. In *International Conference on Learning Representations*, volume 2025, pp. 33944–33963, 2025.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *International Conference on Learning Representations*, volume 2024, pp. 23967–24001, 2024.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 13484–13508, 2023.

Zhexuan Wang, Xuebo Liu, Li Wang, Zifei Shan, Yutong Wang, Zhenxi Song, and Min Zhang. Maspo: Joint prompt optimization for llm-based multi-agent systems. *arXiv preprint arXiv:2605.06623*, 2026.

Bin Wu, Haoran Xu, Xiang Zhuang, Zonghao Chen, Zhu Li, Emine Yilmaz, and Qiang Zhang. From interaction trajectories to prompt rules: Credit assignment for multi-agent prompt optimization. 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.

Yihan Xia, Taotao Wang, Shengli Zhang, Zhangyuhua Weng, Bin Cao, and Soung Chang Liew. Hivemind: Contribution-guided online prompt optimization of llm multi-agent systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29767–29774, 2026.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. Travelplanner: A benchmark for real-world planning with language agents. *arXiv preprint arXiv:2402.01622*, 2024.

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.

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. *arXiv preprint arXiv:2602.03794*, 2026.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 2369–2380, 2018.

Junjie Ye, Zhengyin Du, Xuesong Yao, Weijian Lin, Yufei Xu, Zehui Chen, Zaiyuan Wang, Sining Zhu, Zhiheng Xi, Siyu Yuan, et al. Toolhop: A query-driven benchmark for evaluating large language models in multi-hop tool use. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2995–3021, 2025.

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.

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

Zheyuan Zhang, Lin Ge, Hongjiang Li, Weicheng Zhu, Chuxu Zhang, and Yanfang Ye. Mapro: Recasting multi-agent prompt optimization as maximum a posteriori inference. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 4458–4480, 2026.

Jiaxing Zhao, Hongbin Xie, Yuzhen Lei, Xuan Song, Zhuoran Shi, Lianxin Li, Haoran Zhang, et al. Connecting the dots: A chain-of-collaboration prompting framework for llm agents. *arXiv preprint arXiv:2505.10936*, 2025.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Arik. Multi-agent design: Optimizing agents with better prompts and topologies. In *International Conference on Learning Representations*, volume 2026, pp. 15844–15872, 2026.

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.

Kaijie Zhu, Qinlin Zhao, Hao Chen, Jindong Wang, and Xing Xie. Promptbench: A unified library for evaluation of large language models. *Journal of Machine Learning Research*, 25(254):1–22, 2024.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Daisy Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, et al. Multiagentbench: Evaluating the collaboration and competition of llm agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8580–8622, 2025.

APPENDIX

Contents

A Related Work **15**

B Supplementary Results **17**

B.1 Workflow Topology 17

B.2 Communication Protocol 18

B.3 Team Size 19

B.4 Ablation of Prompt Optimizers 20

B.5 Ablation of Task Model 21

C Benchmark Details **21**

C.1 Frameworks 21

C.2 Task Datasets 22

C.3 Communication Formats 23

C.4 Prompt Optimizers 24

C.5 Models 25

D Prompt Examples **26**

D.1 Meta Prompt 26

D.2 Initial and Optimized System Prompt 26

A RELATED WORK

Prompt Optimization for Single LLM. Prompt optimization improves LLM performance without updating model weights; see (Ramnath et al., 2025; Chang et al., 2024) for detailed reviews. Prompts are typically categorized into system (hard) prompts as discrete text instructions and soft prompts as continuous embeddings (Chang et al., 2024). While both have shown effectiveness in single-agent settings, we focus on system prompts—the discrete instructions that specify each agent's role—due to their interpretability and direct role in specifying agent behavior in MAS.

Most system prompt optimization methods can be viewed as searching over a discrete instruction space (Chang et al., 2024). Existing approaches fall into three categories: (1) sampling-based methods that generate and select candidate prompts using task feedback, including self-generated methods (Wang et al., 2023), LLM-as-optimizer approaches such as APE (Zhou et al., 2022) and OPRO (Yang et al., 2024), planning-based methods such as PromptAgent (Wang et al., 2024), and evolutionary methods such as EvoPrompt (Guo et al., 2024) and PromptBreeder (Fernando et al., 2023); (2) feedback-based methods that leverage directional signals such as reinforcement-learning rewards (Deng et al., 2022), textual gradients (Pryzant et al., 2023; Yuksekgonul et al., 2024), or self-reflection (Madaan et al., 2023; Shinn et al., 2023); and (3) editing-based methods that refine prompts through local operations such as insertion, deletion, or paraphrasing (Prasad et al., 2023). These techniques are also integrated into broader frameworks such as DSPy (Khattab et al., 2023), which optimizes instructions within multi-stage LLM programs via algorithms such as MIPROv2 (Opsahl-Ong et al., 2024). In this work, instead of single-agent, we focus on prompt optimization for multi-agent LLM systems, where it remains unclear whether single-agent gains transfer. To investigate, we evaluate when and how much prompt optimization improves MAS performance across a broad range of setups varying in task, workflow, communication protocol, team size, and different prompt optimizers.

Prompt Optimization in Multi-Agent LLM Systems. Many influential MAS still rely on manually designed system prompts, specifying roles in CAMEL, MetaGPT and ChatDev (Li et al., 2023a; Hong et al., 2024; Qian et al., 2024); specifying conversations and coordination patterns in AutoGen, CrewAI, and AgentVerse (Wu et al., 2023; CrewAI Inc., 2026; Chen et al., 2024c); instantiating collaboration patterns, such as debate (Du et al., 2023; Liang et al., 2024), mixture-style aggregation (Wang et al., 2025), or consensus among diverse LLMs (Chen et al., 2024a). Recently, automatic prompt optimization for MAS has attracted growing attention and shown substantial progress. Many existing methods generate new prompts from diverse feedback sources, including failure attributions and identified underperforming agents (Shen et al., 2025a; Li et al., 2026), judge feedback (Herrera et al., 2026), interaction trajectories (Wu et al., 2026), as well as multi-resolution signals spanning the agent itself, its neighborhood, and global contexts (Wang et al., 2026); while other work searches over a fixed, pre-generated set of candidate prompts rather than generating new ones (Hong et al., 2026). Beyond these directions, researchers have also studied MAS prompt optimization for domain-specific problems (Xia et al., 2026) and incorporated richer MAS-configuration information into the optimization process (Zhang et al., 2026). Recognizing that system prompts are tightly coupled with other design choices, another line of work jointly optimizes prompts with orchestration components such as communication topology and hierarchical planning (Zhao et al., 2025; Zhou et al., 2026).

In this work, we ask a more foundational question: *within a given MAS configuration, how much headroom does system prompt optimization actually offer, and how does this headroom vary across configurations?* We address this through a benchmark-driven study that quantifies how much system-prompt optimization helps across a diverse set of fixed MAS configurations. Specifically, we evaluate the gains—relative to default prompts—of eight prompt optimizers, including multi-agent extensions of state-of-the-art single-agent methods and optimizers designed specifically for MAS. We view these findings as an empirical roadmap that can guide the design of future prompt optimization algorithms—and, more broadly, the design of MAS itself.

Benchmarks for MAS and prompt optimization. Existing work on evaluating and analyzing LLM-based multi-agent systems (MAS) spans three complementary perspectives: task benchmarks, diagnostic tools, and studies of MAS design choices. First, general task benchmarks provide broad testbeds for agents' different abilities within a MAS. Examples include MultiAgentBench (Zhu et al., 2025) for coordination quality, BFCL (Patil et al., 2025) for function calling, AppWorld (Trivedi et al., 2024) for interactive app operation, GAIA (Mialon et al., 2024) for general assistant tasks, TravelPlanner (Xie et al., 2024) for multi-constraint planning, and SWE-bench (Jimenez et al., 2024) for software repair. Second, diagnostic and debugging work addresses the interpretability gap by examining why MAS executions succeed or fail. MAST proposes a taxonomy of failure modes with an annotated trace dataset (Cemri et al., 2026); AutoGen Studio and AGDebugger enable interactive inspection and steering of multi-agent conversations (Dibia et al., 2024; Epperson et al., 2025); and failure-attribution work identifies which agent and step caused a task failure (Zhang et al., 2025). Third, a line of work studies how different MAS configuration choices influence overall performance, including workflow topology (Kim et al., 2025; Shen et al., 2025b), agent diversity (Yang et al., 2026), and team size (Kim et al., 2025; Qian et al., 2025). We follow this third line, but focus on a critical yet underexplored component—the system prompt—studying how much optimizing it can improve MAS performance under a fixed surrounding configuration. Furthermore, although extensive benchmarks exist for evaluating prompt optimization in single-agent setting (Zhu et al., 2024), no comparable benchmark is available for MAS yet. To fill this gap, we introduce a systematic benchmark MAS-PromptBench that measures the gains from MAS prompt optimization across a diverse set of configurations, encompassing different tasks, workflow topologies, team sizes, communication protocols, and prompt optimizers.

B SUPPLEMENTARY RESULTS

B.1 WORKFLOW TOPOLOGY

Table 3: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO for five workflow topologies on nine tasks. Each cell reports baseline / optimized performance, followed by the signed change Δ in percentage points, with the standard deviation of Δ across three optimizer seeds as a subscript. Blue indicates improvement, orange indicates regression, and gray indicates no change.

Dataset	Topology				
	Single	Independent	Sequential	Centralized	Decentralized
			MAS-GEPA		
GPQA	76.3 / 72.3 $-4.0_{\pm 3.5}$	77.3 / 77.3 $0.0_{\pm 2.6}$	72.7 / 72.7 $0.0_{\pm 0.0}$	75.3 / 75.3 $0.0_{\pm 0.0}$	76.7 / 78.0 $+1.3_{\pm 1.5}$
HotpotQA	55.0 / 56.3 $+1.3_{\pm 2.3}$	39.3 / 50.7 $+11.3_{\pm 3.8}$	50.0 / 50.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	55.0 / 55.0 $0.0_{\pm 0.0}$
MATH	67.7 / 71.7 $+4.0_{\pm 3.5}$	55.7 / 61.7 $+6.0_{\pm 7.2}$	69.7 / 75.3 $+5.7_{\pm 1.5}$	69.0 / 71.0 $+2.0_{\pm 4.4}$	78.7 / 80.3 $+1.7_{\pm 2.9}$
LiveCodeBench	68.0 / 84.0 $+16.0_{\pm 6.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	90.7 / 96.0 $+5.3_{\pm 3.1}$	92.0 / 90.0 $-2.0_{\pm 2.0}$	86.0 / 84.0 $-2.0_{\pm 3.5}$
APPS	60.0 / 60.0 $0.0_{\pm 0.0}$	74.0 / 74.0 $0.0_{\pm 0.0}$	70.0 / 70.0 $0.0_{\pm 0.0}$	74.0 / 68.0 $-6.0_{\pm 6.9}$	78.0 / 78.0 $0.0_{\pm 0.0}$
SWE-Bench	40.0 / 40.0 $0.0_{\pm 0.0}$	56.7 / 56.7 $0.0_{\pm 0.0}$	36.7 / 43.3 $+6.7_{\pm 6.7}$	73.3 / 66.7 $-6.7_{\pm 3.3}$	60.0 / 60.0 $0.0_{\pm 0.0}$
BFCL	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	43.3 / 74.0 $+30.7_{\pm 18.0}$	66.0 / 81.7 $+15.7_{\pm 13.3}$	81.0 / 83.3 $+2.3_{\pm 3.2}$
ToolHop	66.0 / 65.0 $-1.0_{\pm 1.7}$	63.7 / 64.0 $+0.3_{\pm 0.6}$	70.7 / 69.0 $-1.7_{\pm 2.1}$	70.0 / 69.7 $-0.3_{\pm 0.6}$	67.7 / 69.0 $+1.3_{\pm 4.0}$
API-Bank	78.7 / 76.3 $-2.3_{\pm 4.0}$	76.0 / 81.0 $+5.0_{\pm 1.7}$	66.0 / 73.3 $+7.3_{\pm 2.1}$	75.3 / 74.3 $-1.0_{\pm 1.0}$	68.7 / 72.0 $+3.3_{\pm 5.8}$
			MAS-MIPRO		
GPQA	78.7 / 78.7 $0.0_{\pm 0.0}$	78.0 / 76.7 $-1.3_{\pm 4.0}$	72.7 / 79.0 $+6.3_{\pm 2.3}$	75.3 / 77.0 $+1.7_{\pm 3.8}$	76.7 / 78.3 $+1.7_{\pm 2.9}$
HotpotQA	54.7 / 56.0 $+1.3_{\pm 1.2}$	53.1 / 53.1 $0.0_{\pm 0.0}$	51.0 / 51.0 $0.0_{\pm 1.0}$	51.3 / 51.3 $0.0_{\pm 0.0}$	55.0 / 56.7 $+1.7_{\pm 2.9}$
MATH	67.7 / 73.3 $+5.7_{\pm 4.5}$	74.3 / 75.7 $+1.3_{\pm 2.3}$	69.7 / 77.3 $+7.7_{\pm 7.1}$	69.0 / 74.7 $+5.7_{\pm 6.7}$	80.3 / 81.7 $+1.3_{\pm 2.3}$
LiveCodeBench	84.0 / 84.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	88.0 / 92.0 $+4.0_{\pm 3.5}$
APPS	64.0 / 78.0 $+14.0_{\pm 2.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$	68.0 / 68.0 $0.0_{\pm 0.0}$	78.0 / 78.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$
SWE-Bench	42.2 / 45.6 $+3.3_{\pm 3.3}$	46.7 / 56.7 $+10.0_{\pm 3.3}$	36.7 / 45.6 $+8.9_{\pm 1.9}$	53.3 / 53.3 $0.0_{\pm 0.0}$	56.7 / 56.7 $0.0_{\pm 0.0}$
BFCL	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	44.3 / 83.3 $+39.0_{\pm 1.7}$	67.0 / 86.0 $+19.0_{\pm 5.0}$	81.3 / 81.3 $0.0_{\pm 0.0}$
ToolHop	68.0 / 68.0 $0.0_{\pm 0.0}$	64.7 / 64.7 $0.0_{\pm 0.0}$	69.3 / 69.3 $0.0_{\pm 3.0}$	71.0 / 70.3 $-0.7_{\pm 1.2}$	66.3 / 66.3 $0.0_{\pm 0.0}$
API-Bank	79.3 / 81.7 $+2.3_{\pm 2.9}$	76.0 / 76.7 $+0.7_{\pm 1.2}$	67.3 / 72.7 $+5.3_{\pm 5.5}$	74.3 / 78.3 $+4.0_{\pm 3.6}$	67.7 / 70.0 $+2.3_{\pm 4.0}$
			MAPRO		
GPQA	—	78.3 / 78.0 $-0.3_{\pm 0.6}$	72.7 / 76.7 $+4.0_{\pm 3.6}$	75.3 / 75.3 $0.0_{\pm 0.0}$	76.7 / 76.7 $0.0_{\pm 0.0}$
HotpotQA	—	46.7 / 46.7 $0.0_{\pm 0.0}$	50.0 / 49.7 $-0.3_{\pm 2.5}$	52.0 / 52.3 $+0.3_{\pm 0.6}$	55.0 / 55.0 $0.0_{\pm 0.0}$
MATH	—	45.3 / 47.3 $+2.0_{\pm 3.5}$	69.7 / 70.7 $+1.0_{\pm 4.6}$	69.0 / 72.7 $+3.7_{\pm 4.0}$	78.3 / 78.3 $0.0_{\pm 0.0}$
LiveCodeBench	—	90.0 / 90.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	92.0 / 92.0 $0.0_{\pm 0.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$
APPS	—	74.0 / 74.0 $0.0_{\pm 0.0}$	78.0 / 72.0 $-6.0_{\pm 0.0}$	70.0 / 70.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$
SWE-Bench	—	60.0 / 56.7 $-3.3_{\pm 0.0}$	30.0 / 40.0 $+10.0_{\pm 3.3}$	63.3 / 63.3 $0.0_{\pm 0.0}$	70.0 / 63.3 $-6.7_{\pm 6.7}$
BFCL	—	54.0 / 54.0 $0.0_{\pm 0.0}$	44.7 / 46.7 $+2.0_{\pm 3.5}$	65.0 / 75.7 $+10.7_{\pm 0.6}$	82.7 / 84.0 $+1.3_{\pm 2.3}$
ToolHop	—	64.7 / 65.3 $+0.7_{\pm 1.2}$	70.7 / 70.3 $-0.3_{\pm 0.6}$	69.0 / 71.0 $+2.0_{\pm 3.5}$	67.0 / 68.7 $+1.7_{\pm 2.9}$
API-Bank	—	76.3 / 76.3 $0.0_{\pm 0.0}$	67.3 / 67.3 $0.0_{\pm 0.0}$	75.3 / 75.3 $0.0_{\pm 0.0}$	68.0 / 68.0 $0.0_{\pm 0.0}$
			MASPO		
GPQA	—	78.3 / 78.7 $+0.3_{\pm 0.6}$	72.7 / 77.0 $+4.3_{\pm 0.6}$	75.3 / 75.3 $0.0_{\pm 0.0}$	76.7 / 78.0 $+1.3_{\pm 1.2}$
HotpotQA	—	44.2 / 49.2 $+5.0_{\pm 4.6}$	50.0 / 51.7 $+1.7_{\pm 1.5}$	54.0 / 53.3 $-0.7_{\pm 1.2}$	55.0 / 57.3 $+2.3_{\pm 4.0}$
MATH	—	45.0 / 45.0 $0.0_{\pm 0.0}$	69.7 / 69.7 $0.0_{\pm 0.0}$	69.0 / 71.0 $+2.0_{\pm 3.5}$	75.7 / 75.7 $0.0_{\pm 0.0}$
LiveCodeBench	—	90.0 / 90.0 $0.0_{\pm 0.0}$	94.0 / 94.0 $0.0_{\pm 0.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$
APPS	—	78.0 / 76.0 $-2.0_{\pm 2.0}$	78.0 / 80.0 $+2.0_{\pm 4.0}$	76.0 / 74.0 $-2.0_{\pm 2.0}$	80.7 / 82.7 $+2.0_{\pm 0.0}$
SWE-Bench	—	46.7 / 50.0 $+3.3_{\pm 3.3}$	23.3 / 31.1 $+7.8_{\pm 1.9}$	56.7 / 56.7 $0.0_{\pm 0.0}$	63.3 / 63.3 $0.0_{\pm 0.0}$
BFCL	—	54.0 / 54.3 $+0.3_{\pm 0.6}$	47.0 / 71.7 $+24.7_{\pm 2.1}$	63.3 / 78.3 $+15.0_{\pm 2.6}$	82.0 / 85.3 $+3.3_{\pm 2.9}$
ToolHop	—	64.0 / 65.7 $+1.7_{\pm 1.5}$	69.0 / 70.7 $+1.7_{\pm 2.1}$	69.7 / 69.7 $0.0_{\pm 0.0}$	67.0 / 69.3 $+2.3_{\pm 2.1}$
API-Bank	—	76.0 / 77.7 $+1.7_{\pm 2.9}$	66.7 / 66.7 $0.0_{\pm 2.0}$	75.7 / 73.7 $-2.0_{\pm 3.5}$	68.0 / 78.0 $+10.0_{\pm 5.0}$

B.2 Communication Protocol

Table 4: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO across communication protocols on HotpotQA, LiveCodeBench, and BFCL. Each cell reports baseline / optimized performance, followed by the signed change Δ in percentage points, with the standard deviation of Δ across three optimizer seeds as a subscript. Blue indicates improvement, orange indicates regression, and gray indicates no change.

Topology	Dataset	Communication protocol		
		Freeform	Semi-structured	Structured
		MAS-GEPA		
Independent	HotpotQA	39.3 / 50.7 $+11.3_{\pm 3.8}$	39.3 / 50.0 $+10.7_{\pm 1.2}$	34.7 / 45.7 $+11.0_{\pm 11.8}$
	LiveCodeBench	86.0 / 86.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$	84.0 / 89.3 $+5.3_{\pm 6.1}$
	BFCL	54.0 / 54.0 $0.0_{\pm 0.0}$	50.0 / 52.3 $+2.3_{\pm 2.1}$	53.3 / 55.3 $+2.0_{\pm 1.0}$
Sequential	HotpotQA	50.0 / 50.0 $0.0_{\pm 0.0}$	45.0 / 48.0 $+3.0_{\pm 6.1}$	44.7 / 48.3 $+3.7_{\pm 0.6}$
	LiveCodeBench	90.7 / 96.0 $+5.3_{\pm 3.1}$	78.0 / 78.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$
	BFCL	43.3 / 74.0 $+30.7_{\pm 18.0}$	61.3 / 76.7 $+15.3_{\pm 13.3}$	59.7 / 76.0 $+16.3_{\pm 6.7}$
Centralized	HotpotQA	54.0 / 54.0 $0.0_{\pm 0.0}$	52.7 / 51.3 $-1.3_{\pm 1.2}$	53.3 / 53.3 $0.0_{\pm 0.0}$
	LiveCodeBench	92.0 / 90.0 $-2.0_{\pm 2.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$
	BFCL	66.0 / 81.7 $+15.7_{\pm 13.3}$	85.7 / 85.0 $-0.7_{\pm 2.1}$	90.0 / 90.7 $+0.7_{\pm 1.2}$
Decentralized	HotpotQA	55.0 / 55.0 $0.0_{\pm 0.0}$	55.3 / 55.7 $+0.3_{\pm 2.5}$	54.7 / 56.0 $+1.3_{\pm 2.3}$
	LiveCodeBench	86.0 / 84.0 $-2.0_{\pm 3.5}$	90.0 / 86.0 $-4.0_{\pm 2.0}$	86.0 / 88.0 $+2.0_{\pm 4.0}$
	BFCL	81.0 / 83.3 $+2.3_{\pm 3.2}$	85.0 / 86.7 $+1.7_{\pm 2.1}$	85.0 / 85.3 $+0.3_{\pm 0.6}$
		MAS-MIPRO		
Independent	HotpotQA	53.1 / 53.1 $0.0_{\pm 0.0}$	49.7 / 55.0 $+5.3_{\pm 9.2}$	50.7 / 50.7 $0.0_{\pm 0.0}$
	LiveCodeBench	86.0 / 86.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	84.0 / 86.0 $+2.0_{\pm 4.0}$
	BFCL	54.0 / 54.0 $0.0_{\pm 0.0}$	50.3 / 51.7 $+1.3_{\pm 2.3}$	52.7 / 53.0 $+0.3_{\pm 0.6}$
Sequential	HotpotQA	51.0 / 51.0 $0.0_{\pm 1.0}$	45.0 / 47.7 $+2.7_{\pm 2.3}$	44.7 / 48.3 $+3.7_{\pm 3.2}$
	LiveCodeBench	90.0 / 90.0 $0.0_{\pm 0.0}$	84.0 / 80.0 $-4.0_{\pm 3.5}$	92.0 / 92.0 $0.0_{\pm 0.0}$
	BFCL	44.3 / 83.3 $+39.0_{\pm 1.7}$	60.3 / 80.0 $+19.7_{\pm 6.8}$	59.7 / 84.7 $+25.0_{\pm 2.0}$
Centralized	HotpotQA	51.3 / 51.3 $0.0_{\pm 0.0}$	54.3 / 53.7 $-0.7_{\pm 1.2}$	54.7 / 54.7 $0.0_{\pm 0.0}$
	LiveCodeBench	86.0 / 86.0 $0.0_{\pm 0.0}$	82.0 / 90.0 $+8.0_{\pm 2.0}$	80.0 / 80.0 $0.0_{\pm 0.0}$
	BFCL	67.0 / 86.0 $+19.0_{\pm 5.0}$	88.3 / 87.7 $-0.7_{\pm 1.2}$	89.3 / 89.0 $-0.3_{\pm 0.6}$
Decentralized	HotpotQA	55.0 / 56.7 $+1.7_{\pm 2.9}$	55.3 / 55.3 $0.0_{\pm 1.0}$	54.7 / 55.0 $+0.3_{\pm 0.6}$
	LiveCodeBench	88.0 / 92.0 $+4.0_{\pm 3.5}$	86.0 / 86.0 $0.0_{\pm 0.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$
	BFCL	81.3 / 81.3 $0.0_{\pm 0.0}$	84.7 / 86.0 $+1.3_{\pm 2.3}$	84.7 / 84.7 $0.0_{\pm 0.0}$
		MAPRO		
Independent	HotpotQA	46.7 / 46.7 $0.0_{\pm 0.0}$	35.1 / 49.1 $+14.0_{\pm 12.2}$	44.2 / 44.2 $0.0_{\pm 0.0}$
	LiveCodeBench	90.0 / 90.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$
	BFCL	54.0 / 54.0 $0.0_{\pm 0.0}$	50.0 / 50.0 $0.0_{\pm 0.0}$	53.3 / 53.3 $0.0_{\pm 0.0}$
Sequential	HotpotQA	50.0 / 49.7 $-0.3_{\pm 2.5}$	45.0 / 50.3 $+5.3_{\pm 1.2}$	44.7 / 49.7 $+5.0_{\pm 6.2}$
	LiveCodeBench	90.0 / 90.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$
	BFCL	44.7 / 46.7 $+2.0_{\pm 3.5}$	64.3 / 67.3 $+3.0_{\pm 7.0}$	60.3 / 62.3 $+2.0_{\pm 6.2}$
Centralized	HotpotQA	52.0 / 52.3 $+0.3_{\pm 0.6}$	53.0 / 54.0 $+1.0_{\pm 1.7}$	54.7 / 54.7 $0.0_{\pm 0.0}$
	LiveCodeBench	92.0 / 92.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$	70.0 / 70.0 $0.0_{\pm 0.0}$
	BFCL	65.0 / 75.7 $+10.7_{\pm 0.6}$	85.7 / 83.7 $-2.0_{\pm 3.5}$	89.0 / 89.0 $0.0_{\pm 0.0}$
Decentralized	HotpotQA	55.0 / 55.0 $0.0_{\pm 0.0}$	55.3 / 55.3 $0.0_{\pm 0.0}$	54.7 / 54.7 $0.0_{\pm 0.0}$
	LiveCodeBench	88.0 / 88.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$
	BFCL	82.7 / 84.0 $+1.3_{\pm 2.3}$	84.7 / 85.7 $+1.0_{\pm 1.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$
		MASPO		
Independent	HotpotQA	44.2 / 49.2 $+5.0_{\pm 4.6}$	42.4 / 44.4 $+2.0_{\pm 5.3}$	38.0 / 45.0 $+7.0_{\pm 6.6}$
	LiveCodeBench	90.0 / 90.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$
	BFCL	54.0 / 54.3 $+0.3_{\pm 0.6}$	50.0 / 52.0 $+2.0_{\pm 1.7}$	53.3 / 54.0 $+0.7_{\pm 1.2}$
Sequential	HotpotQA	50.0 / 51.7 $+1.7_{\pm 1.5}$	45.0 / 46.7 $+1.7_{\pm 1.5}$	44.7 / 53.3 $+8.7_{\pm 3.5}$
	LiveCodeBench	94.0 / 94.0 $0.0_{\pm 0.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$
	BFCL	47.0 / 71.7 $+24.7_{\pm 2.1}$	63.3 / 81.0 $+17.7_{\pm 4.2}$	59.0 / 75.0 $+16.0_{\pm 13.9}$
Centralized	HotpotQA	54.0 / 53.3 $-0.7_{\pm 1.2}$	52.7 / 53.0 $+0.3_{\pm 0.6}$	53.3 / 55.7 $+2.3_{\pm 4.0}$
	LiveCodeBench	88.0 / 88.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$
	BFCL	63.3 / 78.3 $+15.0_{\pm 2.6}$	86.0 / 89.3 $+3.3_{\pm 3.1}$	89.0 / 89.0 $0.0_{\pm 0.0}$
Decentralized	HotpotQA	55.0 / 57.3 $+2.3_{\pm 4.0}$	55.3 / 55.3 $0.0_{\pm 0.0}$	54.7 / 55.3 $+0.7_{\pm 1.2}$
	LiveCodeBench	86.0 / 86.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$
	BFCL	82.0 / 85.3 $+3.3_{\pm 2.9}$	85.3 / 84.3 $-1.0_{\pm 1.0}$	85.3 / 85.3 $0.0_{\pm 0.0}$

B.3 Team Size

Table 5: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO across diverse team sizes on HotpotQA, LiveCodeBench, and BFCL. Each cell reports baseline / optimized performance, followed by the signed change Δ in percentage points, with the standard deviation of Δ across three optimizer seeds as a subscript. Blue indicates improvement, orange indicates regression, and gray indicates no change.

Method	Dataset	Topology	Number of agents		
			n=4	n=8	n=10
MAS-GEPA	HotpotQA	Independent	39.3 / 50.7 $+11.3_{\pm 3.8}$	32.3 / 32.3 $0.0_{\pm 0.0}$	34.3 / 44.0 $+9.7_{\pm 2.1}$
		Sequential	50.0 / 50.0 $0.0_{\pm 0.0}$	50.0 / 50.3 $+0.3_{\pm 0.6}$	50.0 / 51.7 $+1.7_{\pm 2.9}$
		Centralized	54.0 / 54.0 $0.0_{\pm 0.0}$	54.7 / 54.7 $0.0_{\pm 0.0}$	52.7 / 53.0 $+0.3_{\pm 0.6}$
		Decentralized	55.0 / 55.0 $0.0_{\pm 0.0}$	54.0 / 56.3 $+2.3_{\pm 2.5}$	54.7 / 55.3 $+0.7_{\pm 1.2}$
	LiveCodeBench	Independent	86.0 / 86.0 $0.0_{\pm 0.0}$	82.7 / 82.0 $-0.7_{\pm 4.6}$	86.0 / 88.0 $+2.0_{\pm 6.0}$
		Sequential	90.7 / 96.0 $+5.3_{\pm 3.1}$	88.0 / 88.0 $0.0_{\pm 0.0}$	96.0 / 96.0 $0.0_{\pm 0.0}$
		Centralized	92.0 / 90.0 $-2.0_{\pm 2.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$	84.0 / 88.0 $+4.0_{\pm 4.0}$
		Decentralized	86.0 / 84.0 $-2.0_{\pm 3.5}$	82.0 / 90.0 $+8.0_{\pm 4.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$
	BFCL	Independent	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
		Sequential	43.3 / 74.0 $+30.7_{\pm 18.0}$	43.3 / 62.0 $+18.7_{\pm 9.3}$	49.7 / 52.0 $+2.3_{\pm 4.2}$
		Centralized	66.0 / 81.7 $+15.7_{\pm 13.3}$	88.3 / 88.3 $0.0_{\pm 0.0}$	88.7 / 88.3 $-0.3_{\pm 0.6}$
		Decentralized	81.0 / 83.3 $+2.3_{\pm 3.2}$	84.0 / 84.0 $0.0_{\pm 0.0}$	83.7 / 83.7 $0.0_{\pm 0.0}$
MAS-MIPRO	HotpotQA	Independent	53.1 / 53.1 $0.0_{\pm 0.0}$	43.2 / 47.2 $+4.0_{\pm 3.5}$	41.7 / 47.1 $+5.3_{\pm 5.0}$
		Sequential	51.0 / 51.0 $0.0_{\pm 1.0}$	50.0 / 54.7 $+4.7_{\pm 4.2}$	50.0 / 52.7 $+2.7_{\pm 2.5}$
		Centralized	51.3 / 51.3 $0.0_{\pm 0.0}$	55.3 / 55.7 $+0.3_{\pm 0.6}$	52.7 / 51.3 $-1.3_{\pm 1.5}$
		Decentralized	55.0 / 56.7 $+1.7_{\pm 2.9}$	54.7 / 55.7 $+1.0_{\pm 1.7}$	55.0 / 55.3 $+0.3_{\pm 0.6}$
	LiveCodeBench	Independent	86.0 / 86.0 $0.0_{\pm 0.0}$	85.3 / 86.0 $+0.7_{\pm 4.2}$	90.0 / 88.0 $-2.0_{\pm 2.0}$
		Sequential	90.0 / 90.0 $0.0_{\pm 0.0}$	84.0 / 86.0 $+2.0_{\pm 9.2}$	82.0 / 82.0 $0.0_{\pm 0.0}$
		Centralized	86.0 / 86.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$	88.7 / 90.7 $+2.0_{\pm 2.0}$
		Decentralized	88.0 / 92.0 $+4.0_{\pm 3.5}$	86.0 / 86.0 $0.0_{\pm 0.0}$	94.0 / 94.0 $0.0_{\pm 0.0}$
	BFCL	Independent	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
		Sequential	44.3 / 83.3 $+39.0_{\pm 1.7}$	46.7 / 84.3 $+37.7_{\pm 4.2}$	46.3 / 86.7 $+40.3_{\pm 6.7}$
		Centralized	67.0 / 86.0 $+19.0_{\pm 5.0}$	88.3 / 89.0 $+0.7_{\pm 1.2}$	87.7 / 88.0 $+0.3_{\pm 0.6}$
		Decentralized	81.3 / 81.3 $0.0_{\pm 0.0}$	83.0 / 83.7 $+0.7_{\pm 1.2}$	82.3 / 83.0 $+0.7_{\pm 1.2}$
MAPRO	HotpotQA	Independent	46.7 / 46.7 $0.0_{\pm 0.0}$	43.4 / 43.4 $0.0_{\pm 0.0}$	48.9 / 48.9 $0.0_{\pm 0.0}$
		Sequential	50.0 / 49.7 $-0.3_{\pm 2.5}$	50.0 / 51.0 $+1.0_{\pm 1.7}$	50.0 / 49.7 $-0.3_{\pm 0.6}$
		Centralized	52.0 / 52.3 $+0.3_{\pm 0.6}$	55.0 / 55.0 $0.0_{\pm 0.0}$	52.7 / 52.7 $0.0_{\pm 0.0}$
		Decentralized	55.0 / 55.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	55.0 / 55.0 $0.0_{\pm 0.0}$
	LiveCodeBench	Independent	90.0 / 90.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$	88.7 / 84.7 $-4.0_{\pm 7.2}$
		Sequential	90.0 / 90.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$
		Centralized	92.0 / 92.0 $0.0_{\pm 0.0}$	86.7 / 86.0 $-0.7_{\pm 2.3}$	88.0 / 88.0 $0.0_{\pm 0.0}$
		Decentralized	88.0 / 88.0 $0.0_{\pm 0.0}$	84.0 / 84.0 $0.0_{\pm 0.0}$	94.0 / 94.0 $0.0_{\pm 0.0}$
	BFCL	Independent	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	53.3 / 53.3 $0.0_{\pm 0.0}$
		Sequential	44.7 / 46.7 $+2.0_{\pm 3.5}$	46.0 / 62.7 $+16.7_{\pm 11.4}$	52.0 / 52.0 $0.0_{\pm 0.0}$
		Centralized	65.0 / 75.7 $+10.7_{\pm 0.6}$	87.7 / 88.0 $+0.3_{\pm 2.5}$	89.0 / 89.0 $0.0_{\pm 0.0}$
		Decentralized	82.7 / 84.0 $+1.3_{\pm 2.3}$	82.3 / 82.3 $0.0_{\pm 0.0}$	83.0 / 83.0 $0.0_{\pm 0.0}$
MASPO	HotpotQA	Independent	44.2 / 49.2 $+5.0_{\pm 4.6}$	37.7 / 38.0 $+0.3_{\pm 2.3}$	30.3 / 40.3 $+10.0_{\pm 7.0}$
		Sequential	50.0 / 51.7 $+1.7_{\pm 1.5}$	50.0 / 48.7 $-1.3_{\pm 2.3}$	49.0 / 50.3 $+1.3_{\pm 2.3}$
		Centralized	54.0 / 53.3 $-0.7_{\pm 1.2}$	54.7 / 54.3 $-0.3_{\pm 0.6}$	52.3 / 53.0 $+0.7_{\pm 1.2}$
		Decentralized	55.0 / 57.3 $+2.3_{\pm 4.0}$	54.0 / 54.7 $+0.7_{\pm 2.1}$	53.7 / 52.3 $-1.3_{\pm 2.3}$
	LiveCodeBench	Independent	90.0 / 90.0 $0.0_{\pm 0.0}$	82.0 / 87.3 $+5.3_{\pm 4.6}$	86.0 / 94.0 $+8.0_{\pm 6.0}$
		Sequential	94.0 / 94.0 $0.0_{\pm 0.0}$	92.0 / 92.0 $0.0_{\pm 0.0}$	92.0 / 90.0 $-2.0_{\pm 0.0}$
		Centralized	88.0 / 88.0 $0.0_{\pm 0.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$	96.0 / 96.0 $0.0_{\pm 0.0}$
		Decentralized	86.0 / 86.0 $0.0_{\pm 0.0}$	82.0 / 82.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$
	BFCL	Independent	54.0 / 54.3 $+0.3_{\pm 0.6}$	54.0 / 54.3 $+0.3_{\pm 0.6}$	53.7 / 54.0 $+0.3_{\pm 0.6}$
		Sequential	47.0 / 71.7 $+24.7_{\pm 2.1}$	44.7 / 54.3 $+9.7_{\pm 16.7}$	52.0 / 56.7 $+4.7_{\pm 8.1}$
		Centralized	63.3 / 78.3 $+15.0_{\pm 2.6}$	88.3 / 88.3 $0.0_{\pm 0.0}$	88.7 / 88.3 $-0.3_{\pm 0.6}$
		Decentralized	82.0 / 85.3 $+3.3_{\pm 2.9}$	82.7 / 82.7 $0.0_{\pm 0.0}$	83.7 / 86.0 $+2.3_{\pm 2.1}$

B.4 Ablation of Prompt Optimizers

Table 6: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, HiveMind, MAPRO, MAMUT-GEPA, MASPO, MASPOB, and TRUCE for five workflow topologies on HotpotQA, LiveCodeBench, and BFCL. Each cell reports baseline / optimized performance, followed by the signed change Δ in percentage points, with the standard deviation of Δ across three optimizer seeds as a subscript. Blue indicates improvement, orange indicates regression, and gray indicates no change.

Method	Topology	Dataset		
		HotpotQA	LiveCodeBench	BFCL
MAS-GEPA	Single	55.0 / 56.3 $+1.3_{\pm 2.3}$	68.0 / 84.0 $+16.0_{\pm 6.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
	Independent	39.3 / 50.7 $+11.3_{\pm 3.8}$	86.0 / 86.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
	Sequential	50.0 / 50.0 $0.0_{\pm 0.0}$	90.7 / 96.0 $+5.3_{\pm 3.1}$	43.3 / 74.0 $+30.7_{\pm 18.0}$
	Centralized	54.0 / 54.0 $0.0_{\pm 0.0}$	92.0 / 90.0 $-2.0_{\pm 2.0}$	66.0 / 81.7 $+15.7_{\pm 13.3}$
	Decentralized	55.0 / 55.0 $0.0_{\pm 0.0}$	86.0 / 84.0 $-2.0_{\pm 3.5}$	81.0 / 83.3 $+2.3_{\pm 3.2}$
MAS-MIPRO	Single	54.7 / 56.0 $+1.3_{\pm 1.2}$	84.0 / 84.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
	Independent	53.1 / 53.1 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
	Sequential	51.0 / 51.0 $0.0_{\pm 1.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	44.3 / 83.3 $+39.0_{\pm 1.7}$
	Centralized	51.3 / 51.3 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	67.0 / 86.0 $+19.0_{\pm 5.0}$
	Decentralized	55.0 / 56.7 $+1.7_{\pm 2.9}$	88.0 / 92.0 $+4.0_{\pm 3.5}$	81.3 / 81.3 $0.0_{\pm 0.0}$
HiveMind	Independent	50.0 / 50.0 $0.0_{\pm 0.0}$	93.3 / 94.0 $+0.7_{\pm 3.1}$	54.3 / 54.3 $0.0_{\pm 0.0}$
	Sequential	50.0 / 49.3 $-0.7_{\pm 1.2}$	92.0 / 90.0 $-2.0_{\pm 2.0}$	43.7 / 50.0 $+6.3_{\pm 4.5}$
	Centralized	52.7 / 52.7 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	70.0 / 73.0 $+3.0_{\pm 7.0}$
	Decentralized	52.7 / 53.7 $+1.0_{\pm 1.7}$	90.0 / 80.0 $-10.0_{\pm 3.5}$	82.3 / 82.3 $0.0_{\pm 0.0}$
MAPRO	Independent	46.7 / 46.7 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
	Sequential	50.0 / 49.7 $-0.3_{\pm 2.5}$	90.0 / 90.0 $0.0_{\pm 0.0}$	44.7 / 46.7 $+2.0_{\pm 3.5}$
	Centralized	52.0 / 52.3 $+0.3_{\pm 0.6}$	92.0 / 92.0 $0.0_{\pm 0.0}$	65.0 / 75.7 $+10.7_{\pm 0.6}$
	Decentralized	55.0 / 55.0 $0.0_{\pm 0.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$	82.7 / 84.0 $+1.3_{\pm 2.3}$
MAMUT-GEPA	Independent	45.1 / 50.7 $+5.7_{\pm 6.7}$	86.0 / 86.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
	Sequential	50.0 / 49.7 $-0.3_{\pm 0.6}$	92.0 / 94.7 $+2.7_{\pm 4.2}$	47.0 / 71.7 $+24.7_{\pm 10.2}$
	Centralized	54.0 / 53.0 $-1.0_{\pm 1.7}$	92.0 / 92.0 $0.0_{\pm 0.0}$	66.0 / 78.3 $+12.3_{\pm 10.8}$
	Decentralized	55.0 / 56.0 $+1.0_{\pm 1.7}$	90.0 / 94.0 $+4.0_{\pm 2.0}$	81.3 / 84.0 $+2.7_{\pm 4.6}$
MASPO	Independent	44.2 / 49.2 $+5.0_{\pm 4.6}$	90.0 / 90.0 $0.0_{\pm 0.0}$	54.0 / 54.3 $+0.3_{\pm 0.6}$
	Sequential	50.0 / 51.7 $+1.7_{\pm 1.5}$	94.0 / 94.0 $0.0_{\pm 0.0}$	47.0 / 71.7 $+24.7_{\pm 2.1}$
	Centralized	54.0 / 53.3 $-0.7_{\pm 1.2}$	88.0 / 88.0 $0.0_{\pm 0.0}$	63.3 / 78.3 $+15.0_{\pm 2.6}$
	Decentralized	55.0 / 57.3 $+2.3_{\pm 4.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	82.0 / 85.3 $+3.3_{\pm 2.9}$
MASPOB	Independent	44.1 / 51.5 $+7.3_{\pm 6.4}$	86.0 / 86.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
	Sequential	50.0 / 49.5 $-0.5_{\pm 0.7}$	88.0 / 88.0 $0.0_{\pm 0.0}$	47.0 / 67.0 $+20.0_{\pm 4.0}$
	Centralized	54.0 / 54.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	67.0 / 71.0 $+4.0_{\pm 7.8}$
	Decentralized	53.0 / 53.3 $+0.3_{\pm 0.6}$	86.0 / 86.7 $+0.7_{\pm 6.4}$	82.0 / 84.0 $+2.0_{\pm 2.6}$
TRUCE	Independent	41.5 / 44.1 $+2.7_{\pm 4.6}$	88.0 / 88.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$
	Sequential	50.0 / 50.0 $0.0_{\pm 0.0}$	91.3 / 93.3 $+2.0_{\pm 4.0}$	45.7 / 50.7 $+5.0_{\pm 8.7}$
	Centralized	54.0 / 52.0 $-2.0_{\pm 3.5}$	88.7 / 89.3 $+0.7_{\pm 11.0}$	67.0 / 66.3 $-0.7_{\pm 12.0}$
	Decentralized	55.0 / 55.3 $+0.3_{\pm 0.6}$	84.0 / 94.0 $+10.0_{\pm 6.0}$	81.7 / 84.3 $+2.7_{\pm 3.1}$

B.5 Ablation of Task Model

Table 7: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO for five workflow topologies on HotpotQA, LiveCodeBench, and BFCL with Qwen3.5-9B as the task model. Each cell reports baseline / optimized performance, followed by the signed change Δ in percentage points, with the standard deviation of Δ across three optimizer seeds as a subscript. Blue indicates improvement, orange indicates regression, and gray indicates no change.

Method	Dataset	Topology				
		Single	Independent	Sequential	Centralized	Decentralized
MAS-GEPA	HotpotQA	55.0 / 56.3 $+1.3_{\pm 2.3}$	39.3 / 50.7 $+11.3_{\pm 3.8}$	50.0 / 50.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	55.0 / 55.0 $0.0_{\pm 0.0}$
	LiveCodeBench	68.0 / 84.0 $+16.0_{\pm 6.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	90.7 / 96.0 $+5.3_{\pm 3.1}$	92.0 / 90.0 $-2.0_{\pm 2.0}$	86.0 / 84.0 $-2.0_{\pm 3.5}$
	BFCL	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	43.3 / 74.0 $+30.7_{\pm 18.0}$	66.0 / 81.7 $+15.7_{\pm 13.3}$	81.0 / 83.3 $+2.3_{\pm 3.2}$
MAS-MIPRO	HotpotQA	54.7 / 56.0 $+1.3_{\pm 1.2}$	53.1 / 53.1 $0.0_{\pm 0.0}$	51.0 / 51.0 $0.0_{\pm 1.0}$	51.3 / 51.3 $0.0_{\pm 0.0}$	55.0 / 56.7 $+1.7_{\pm 2.9}$
	LiveCodeBench	84.0 / 84.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$	88.0 / 92.0 $+4.0_{\pm 3.5}$
	BFCL	54.0 / 54.0 $0.0_{\pm 0.0}$	54.0 / 54.0 $0.0_{\pm 0.0}$	44.3 / 83.3 $+39.0_{\pm 1.7}$	67.0 / 86.0 $+19.0_{\pm 5.0}$	81.3 / 81.3 $0.0_{\pm 0.0}$
MAPRO	HotpotQA	—	46.7 / 46.7 $0.0_{\pm 0.0}$	50.0 / 49.7 $-0.3_{\pm 2.5}$	52.0 / 52.3 $+0.3_{\pm 0.6}$	55.0 / 55.0 $0.0_{\pm 0.0}$
	LiveCodeBench	—	90.0 / 90.0 $0.0_{\pm 0.0}$	90.0 / 90.0 $0.0_{\pm 0.0}$	92.0 / 92.0 $0.0_{\pm 0.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$
	BFCL	—	54.0 / 54.0 $0.0_{\pm 0.0}$	44.7 / 46.7 $+2.0_{\pm 3.5}$	65.0 / 75.7 $+10.7_{\pm 0.6}$	82.7 / 84.0 $+1.3_{\pm 2.3}$
MASPO	HotpotQA	—	44.2 / 49.2 $+5.0_{\pm 4.6}$	50.0 / 51.7 $+1.7_{\pm 1.5}$	54.0 / 53.3 $-0.7_{\pm 1.2}$	55.0 / 57.3 $+2.3_{\pm 4.0}$
	LiveCodeBench	—	90.0 / 90.0 $0.0_{\pm 0.0}$	94.0 / 94.0 $0.0_{\pm 0.0}$	88.0 / 88.0 $0.0_{\pm 0.0}$	86.0 / 86.0 $0.0_{\pm 0.0}$
	BFCL	—	54.0 / 54.3 $+0.3_{\pm 0.6}$	47.0 / 71.7 $+24.7_{\pm 2.1}$	63.3 / 78.3 $+15.0_{\pm 2.6}$	82.0 / 85.3 $+3.3_{\pm 2.9}$

Table 8: Prompt-optimization gains of MAS-GEPA, MAS-MIPRO, MAPRO, and MASPO for five workflow topologies on HotpotQA, LiveCodeBench, and BFCL with Llama-3.1-8B-Instruct as the task model. Each cell reports baseline / optimized performance, followed by the signed change Δ in percentage points, with the standard deviation of Δ across three optimizer seeds as a subscript. Blue indicates improvement, orange indicates regression, and gray indicates no change.

Method	Dataset	Topology				
		Single	Independent	Sequential	Centralized	Decentralized
MAS-GEPA	HotpotQA	34.0 / 32.0 $-2.0_{\pm 1.7}$	29.4 / 37.1 $+7.7_{\pm 5.0}$	41.3 / 40.3 $-1.0_{\pm 1.7}$	28.7 / 29.7 $+1.0_{\pm 1.7}$	33.9 / 32.6 $-1.4_{\pm 3.8}$
	LiveCodeBench	23.0 / 23.0 $0.0_{\pm 0.0}$	28.0 / 26.0 $-2.0_{\pm 2.0}$	19.0 / 14.0 $-5.0_{\pm 7.1}$	18.0 / 18.0 $0.0_{\pm 0.0}$	26.0 / 26.0 $0.0_{\pm 0.0}$
	BFCL	33.6 / 73.5 $+39.9_{\pm 8.3}$	19.7 / 68.3 $+48.7_{\pm 13.4}$	69.7 / 78.7 $+9.0_{\pm 6.2}$	17.3 / 63.0 $+45.7_{\pm 17.6}$	79.3 / 82.9 $+3.7_{\pm 4.0}$
MAS-MIPRO	HotpotQA	32.3 / 32.3 $0.0_{\pm 0.0}$	28.0 / 33.0 $+5.0_{\pm 4.4}$	39.7 / 40.0 $+0.3_{\pm 0.6}$	29.7 / 30.0 $+0.3_{\pm 0.6}$	30.7 / 32.0 $+1.3_{\pm 2.3}$
	LiveCodeBench	24.0 / 24.0 $0.0_{\pm 0.0}$	22.0 / 22.0 $0.0_{\pm 0.0}$	20.0 / 16.0 $-4.0_{\pm 2.0}$	18.0 / 18.0 $0.0_{\pm 0.0}$	26.0 / 24.0 $-2.0_{\pm 6.0}$
	BFCL	33.7 / 72.7 $+39.0_{\pm 4.6}$	20.7 / 64.7 $+44.0_{\pm 4.6}$	64.0 / 82.3 $+18.3_{\pm 4.9}$	15.0 / 71.0 $+56.0_{\pm 6.9}$	78.3 / 80.9 $+2.7_{\pm 3.1}$
MAPRO	HotpotQA	—	33.1 / 32.4 $-0.7_{\pm 1.2}$	39.3 / 37.0 $-2.3_{\pm 2.1}$	28.3 / 28.3 $0.0_{\pm 0.0}$	37.0 / 36.3 $-0.7_{\pm 1.2}$
	LiveCodeBench	—	26.0 / 26.0 $0.0_{\pm 0.0}$	12.0 / 12.0 $0.0_{\pm 0.0}$	14.0 / 14.0 $0.0_{\pm 0.0}$	28.0 / 28.0 $0.0_{\pm 0.0}$
	BFCL	—	22.3 / 41.0 $+18.7_{\pm 16.2}$	70.0 / 72.0 $+2.0_{\pm 3.5}$	12.3 / 11.3 $-1.0_{\pm 5.6}$	80.9 / 84.2 $+3.4_{\pm 3.1}$
MASPO	HotpotQA	—	31.3 / 31.3 $0.0_{\pm 0.0}$	43.0 / 43.0 $0.0_{\pm 0.0}$	28.3 / 30.0 $+1.7_{\pm 2.9}$	34.7 / 34.3 $-0.3_{\pm 1.2}$
	LiveCodeBench	—	24.0 / 24.0 $0.0_{\pm 0.0}$	16.0 / 15.0 $-1.0_{\pm 1.4}$	16.0 / 16.0 $0.0_{\pm 0.0}$	28.0 / 28.0 $0.0_{\pm 0.0}$
	BFCL	—	26.0 / 68.0 $+42.0_{\pm 7.0}$	59.7 / 59.7 $0.0_{\pm 0.0}$	13.0 / 63.3 $+50.3_{\pm 8.1}$	83.2 / 82.5 $-0.7_{\pm 1.2}$

C Benchmark Details

C.1 Frameworks

We instantiate MAS configurations using four public frameworks: LangGraph (LangChain Inc., 2026), CrewAI (CrewAI Inc., 2026), AutoGen (Wu et al., 2023), and OpenAI Agents SDK (OpenAI, 2026). Together, these frameworks span graph-based orchestration, role-based collaboration, conversational multi-agent systems, and production-oriented agent workflows, allowing us to evaluate prompt optimization across diverse execution environments.

LangGraph (LangChain Inc., 2026) is a graph-based framework for building stateful language-agent workflows. Agents are represented as nodes and information flow is defined through directed graph edges, making it well suited for implementing sequential, branching, and cyclic coordination structures. We use LangGraph to instantiate topologies where explicit control over routing and state propagation is required.

CrewAI (CrewAI Inc., 2026) is a role-based multi-agent framework that organizes agents around specialized responsibilities and task delegation. Agents collaborate through predefined roles, goals,

and communication patterns, providing a natural abstraction for workflows that emphasize specialization and hierarchical coordination. We use CrewAI to study how prompt optimization interacts with structured role assignments.

AutoGen (Wu et al., 2023) is a conversational multi-agent framework in which agents interact through iterative message exchange. It provides flexible support for debate, reflection, collaboration, and tool use, making it a common platform for research on LLM-based agent societies. We use AutoGen to instantiate communication-intensive workflows where performance depends heavily on inter-agent interaction.

OpenAI Agents SDK (OpenAI, 2026) is a production-oriented framework for building tool-using agents with tracing, handoffs, and structured execution. The framework provides native support for agent delegation, tool invocation, and workflow monitoring, making it representative of modern agent-engineering practice. We use it to evaluate prompt optimization in realistic agent pipelines that combine reasoning, coordination, and external tool use.

C.2 Task Datasets

We choose benchmarks to cover three main regimes where MAS are commonly used: reasoning, coding, and tool use. This mix lets us test whether prompt optimization helps only on tasks with explicit artifacts, such as code, patches, or function calls, or also on tasks where agents mainly exchange rationales and final answers. For each dataset, we use its native evaluation metric, reported as accuracy, pass rate, or resolve rate depending on the benchmark.

GPQA-Diamond (Rein et al., 2023) is the Diamond subset of GPQA, a graduate-level, Google-proof multiple-choice benchmark written by domain experts in biology, physics, and chemistry. The questions are designed to be difficult even for highly capable language models and resistant to retrieval-based shortcuts. We use it to evaluate scientific reasoning under a constrained multiple-choice format. We report multiple-choice answer accuracy: the model's final selected option is extracted and compared with the gold option, and a prediction is correct only when the selected option exactly matches the reference answer. From the 198 available examples, we use 48 training, 50 validation, and 100 held-out test examples, with no example shared across splits.

HotpotQA (Yang et al., 2018) is a multi-hop question-answering benchmark built from Wikipedia. Answering a question typically requires combining evidence from multiple documents rather than retrieving a single supporting passage. We use it to evaluate evidence integration and multi-step reasoning in collaborative agent workflows. We use SQuAD-style exact match: both prediction and reference answer are normalized by lowercasing, removing punctuation and articles, and standardizing whitespace, and a prediction is correct only if the normalized prediction exactly matches the normalized reference answer. From the 7,405 available examples, we use 150 training, 50 validation, and 100 held-out test examples, with no example shared across splits.

MATH (Hendrycks et al., 2021b) contains competition-level mathematics problems spanning algebra, geometry, number theory, probability, and calculus. Solving these problems often requires long chains of symbolic reasoning and precise intermediate calculations. We use it to evaluate mathematical reasoning. We report math-equivalence accuracy on the extracted final answer: following the benchmark format, we extract the answer from `\boxed{}` when available, otherwise from the final answer span, and count a prediction as correct if the extracted answer is mathematically equivalent to the ground-truth answer. From the 312 available examples, we use 150 training, 50 validation, and 100 held-out test examples, with no example shared across splits.

LiveCodeBench (Jain et al., 2025) is a contamination-resistant coding benchmark built from recent programming-contest problems. Because the tasks are collected after the training cutoff of many language models, they provide a stronger test of generalization than static coding benchmarks. We use it to evaluate code generation under executable test cases. We report all-tests-pass accuracy: the generated program is executed against the benchmark test suite, and an instance is counted as correct only if all hidden test cases pass. From the 1,055 available examples, we use 150 training, 50 validation, and 50 held-out test examples, with no example shared across splits.

APPS (Hendrycks et al., 2021a) evaluates code generation from natural-language programming specifications. The benchmark spans introductory, interview-level, and competition-style programming problems with hidden test cases. We use it to test whether agents can synthesize correct programs

from problem descriptions alone. We also report all-tests-pass accuracy: the generated solution is run against the benchmark test cases, and an instance is correct only when the program passes the full test suite; passing public examples alone is not sufficient. From the 3,000 available examples, we use 150 training, 50 validation, and 50 held-out test examples, with no example shared across splits.

SWE-Bench Verified (Jimenez et al., 2024) is a human-validated subset of SWE-bench built from real GitHub issues and software repositories. Each instance requires understanding an existing codebase, modifying repository files, and generating a patch that resolves the reported issue. We use it to evaluate repository-level software engineering tasks under executable verification. We report resolve rate: the generated patch is applied to the target repository and evaluated with the benchmark's issue-resolution tests, and an instance is counted as resolved only if the patch applies successfully and all required `FAIL_TO_PASS` and `PASS_TO_PASS` tests pass. From the 500 available examples, we use 150 training, 50 validation, and 30 held-out test examples, with no example shared across splits.

BFCL (Patil et al., 2025) evaluates function-calling ability across realistic tool-use settings. Tasks require selecting the correct function and generating valid arguments that satisfy the API specification. We use it to evaluate structured tool invocation and argument generation. We report AST-based function-call correctness: the predicted function call is parsed into an abstract syntax tree and compared with the reference call by function name and argument values, accepting formatting differences that do not change the function call semantics. From the 1,000 available examples, we use 150 training, 50 validation, and 100 held-out test examples, with no example shared across splits.

ToolHop (Ye et al., 2025) evaluates multi-hop tool use, where solving a query requires selecting and composing multiple locally executable tools. The output of one tool often serves as the input to another, creating dependencies across tool calls. We use it to evaluate sequential tool planning and execution. We report answer accuracy: the agent must select and compose the required tools, then return a final answer, and a prediction is correct when the final answer after tool execution matches the benchmark reference answer. From the 995 available examples, we use 150 training, 50 validation, and 100 held-out test examples, with no example shared across splits.

API-Bank (Li et al., 2023b) evaluates tool-augmented dialogue agents in a runnable API environment. Tasks require planning, API retrieval, parameter selection, and API execution within multi-turn interactions. We use it to evaluate end-to-end tool-use behavior in interactive settings. We report API-call accuracy: a prediction is correct when the model selects the correct API and provides the required arguments according to the annotated reference call, evaluating both API retrieval and parameter generation. From the 445 available examples, we use 150 training, 50 validation, and 100 held-out test examples, with no example shared across splits.

C.3 Communication Formats

We compare three inter-agent communication formats using the same HotpotQA example, in which an agent reports evidence for whether Scott Derrickson and Ed Wood share the same nationality. The freeform format provides only the question and answer context, leaving downstream agents to infer which facts are important. The semi-structured format exposes the agent's status, summary, evidence, confidence, next step, entities, reasoning hops, and answer candidate through explicit tags. The structured format encodes the same information as a JSON-style report, making the message easier to parse and validate automatically. Across all three examples, only the communication format changes; the task, topology, team size, agent roles, and scoring rule remain fixed.

HotpotQA—multi-hop comparison

```
Question: Were Scott Derrickson and Ed Wood of the same nationality?
Gold answer: yes
```

Freeform Message

```
researcher: I looked up both people. Scott Derrickson is an American
director (born in Denver, Colorado), and Ed Wood was an American
filmmaker (born in Poughkeepsie, New York). Since both are American,
they share the same nationality--so the answer is yes.
```

Semi-structured Message

```
[STATUS] completed
[SUMMARY] Both individuals are American, so they share the same
nationality.
[EVIDENCE_OR_TESTS]
    • Scott Derrickson (Wikipedia): American director, born in
      Denver, Colorado.
    • Ed Wood (Wikipedia): American filmmaker, born in Poughkeepsie,
      New York.
[CONFIDENCE] high
[NEXT] Hand off to the writer to emit the final yes/no answer.
[ENTITIES] Scott Derrickson, Ed Wood
[HOPS] Find Scott Derrickson's nationality; find Ed Wood's
nationality; compare.
[ANSWER_CANDIDATE] yes
```

Structured Message Example

```
JSON_REPORT:
{
    "status": "completed",
    "summary": "Both individuals are American, so they share
               the same nationality.",
    "confidence": "high",
    "next": "Hand off to the writer to emit the final
           yes/no answer.",
    "payload": {
    "entities": ["Scott Derrickson", "Ed Wood"],
    "hops": [
      "Find Scott Derrickson's nationality",
      "Find Ed Wood's nationality",
      "Compare the two"
    ],
    "evidence": [
      {
        "source": "Scott Derrickson (Wikipedia)",
        "fact": "American director, born in Denver, Colorado"
      },
      {
        "source": "Ed Wood (Wikipedia)",
        "fact": "American filmmaker, born in "
                "Poughkeepsie, New York"
      }
    ],
    "answer_candidate": "yes"
    }
}
END_JSON_REPORT
```

C.4 Prompt Optimizers

Multi-agent extension of GEPA. GEPA (Agrawal et al., 2026) is a state-of-the-art prompt optimization framework originally designed for single-agent LLM systems. It improves prompts through a reflection-based optimization procedure that leverages natural-language feedback to iteratively revise prompts based on execution traces, while keeping the underlying model weights fixed. Specifically, GEPA maintains a pool of candidate prompts. In each optimization iteration, it selects a candidate from the Pareto frontier of the current prompt pool, executes the corresponding system on a minibatch of training tasks, and records the resulting execution traces, including intermediate reasoning steps, tool invocations, tool outputs, and final answers. A feedback function then evaluates each rollout and produces both a scalar task score and textual feedback. Together with the associated execution trace, this information is provided to a reflection model, which analyzes the observed failures and generates

a revised prompt candidate that is added back to the candidate pool. This process repeats until the rollout budget is exhausted.

We extend this reflective prompt-evolution framework to multi-agent LLM systems. GEPA maintains a pool of candidate prompt configurations, each assigning a system prompt to every agent. During each optimization round, GEPA updates agents sequentially, optimizing one agent's prompt at a time while keeping all others fixed. For the selected agent, the reflection model receives the agent's execution trace, the surrounding interaction context, the final team-level outcome, and feedback produced by the evaluation function. Based on this information, it revises only the selected agent's system prompt, leaving the prompts of all other agents unchanged.

For all experiments, we use the training and validation splits of each dataset in Appendix C.2 (150 training and 50 validation examples; 48 training examples for GPQA-Diamond) for each combination of dataset and topology, together with a rollout budget of $B = 600$ in equation 2. At each iteration, GEPA samples a candidate prompt configuration and evaluates it on a minibatch of three training examples; using a round-robin policy, it then reflects on the resulting traces and feedback to revise one agent's system prompt at a time. We disable perfect-score skipping so that optimization continues on easy datasets where minibatches may already score perfectly, and we terminate optimization after five full-validation iterations without improvement.

After optimization, we adopt a conservative prompt-selection strategy at the system level. The optimized multi-agent prompt configuration is used only if it outperforms the original seed configuration on the GEPA validation split; otherwise, the seed configuration is retained. Final benchmark evaluation is performed separately from optimization, and all examples used during prompt optimization are excluded from later evaluation.

Multi-agent extension of MIPRO. MIPRO (Opsahl-Ong et al., 2024) is a prompt optimizer originally proposed for single-agent LM programs, where a program may contain one or multiple LLM modules arranged as a multi-stage pipeline. We directly adapt MIPRO to multi-agent LLM systems by treating each agent as an optimizable module. For each dataset and collaboration topology, we keep the original multi-agent execution engine and inter-agent communication structure unchanged, and optimize only the system prompt of each agent. This allows MIPRO to search over combinations of per-agent prompt candidates using end-to-end task performance as the optimization signal.

For all experiments, MIPRO uses the same training and validation splits as GEPA. For each agent, it proposes three candidate instructions and three bootstrapped demonstration sets of up to four demonstrations each, using no manually labeled examples, and searches over their combinations for three optimization trials. Candidate multi-agent prompt configurations are evaluated on the full validation set rather than minibatches, and both instruction candidates and rendered few-shot demonstrations are selected based on validation performance. After optimization, we adopt the same conservative prompt-selection strategy used for GEPA.

C.5 Models

Table 9 summarizes the models used throughout the benchmark. We use `Qwen/Qwen3.5-9B` as the task model for all benchmark execution and `Qwen/Qwen3.5-122B-A10B-FP8` as the reflection model for prompt optimization. For the task-model ablation in Section 4.5, we replace the task model with `meta-llama/Llama-3.1-8B-Instruct` and keep the reflection model unchanged. This separation follows the design of modern prompt optimizers: the task model executes the benchmark under a given prompt configuration, while the reflection model analyzes failures and proposes prompt updates. All reported results are produced by re-running the benchmark with the resulting optimized prompts on the held-out test split.

Disable Thinking Mode. We disable thinking mode for the task model (Qwen3.5-9B) to maintain a controlled and reproducible evaluation protocol. This avoids differences in hidden reasoning budgets across tasks, topologies, communication protocols, and team sizes, ensuring that comparisons reflect visible agent behavior and coordination rather than variation in model-internal reasoning. Therefore, the reported scores should be interpreted as performance under a controlled agentic protocol rather than the maximum achievable performance of the underlying model.

Table 9: Models used for task execution and prompt optimization, with decoding configurations. All prompt optimizers share the same task and reflection models; decoding values follow the GEPA configuration.

		Task model	Task model (ablation)	Reflection model
Configuration	Model ID	`Qwen/Qwen3.5-9B`	`meta-llama/Llama-3.1-8B-Instruct`	`Qwen/Qwen3.5-122B-A10B-FP8`
	Temperature	0.2	0.2	1.0
	Top-p	0.9	0.9	1.0 (default)
	Decoding seed	0	0	— (unset)
	Max output tokens	32,768	32,768	48,000
	Thinking mode	Disabled	N/A	Enabled

D Prompt Examples

D.1 Meta Prompt

We use a meta prompt to generate role-specific seed prompts for each benchmark and topology. For each agent, we fill its placeholders with the task domain, the topology description, the available tools, and the agent's role and role description.

Meta Prompt Template

```
You are designing a multi-agent system.

DOMAIN:
{DOMAIN_BACKGROUND}

TOPOLOGY:
{TOPOLOGY_DESC}

AVAILABLE TOOLS:
{TOOLS}

Write an initial system prompt for an agent playing the role "{ROLE}"
    in this topology. The agent will work on tasks in the domain
    described above.

Role description: {ROLE_DESC}

The generated system prompt must only reference the tools listed
    above. Do not invent additional tools (web search, file readers,
    code executors, etc.) that are not in the list. If the list is
    "None", the agent must reason directly from the prompt without
    any tool use.

Output only the system prompt text.
```

D.2 Initial and Optimized System Prompt

This section presents representative baseline and optimized system prompts from our experiments. For each example, we report the original role-specific prompt used as the seed and the corresponding prompt selected after system-prompt optimization. These examples show how optimization changes agent instructions by clarifying task requirements, tool-use procedures, output constraints, and coordination behavior, while keeping the benchmark, topology, available tools, number of agents, and aggregation rule fixed.

D.2.1 HOTPOTQA UNDER INDEPENDENT TOPOLOGY

Baseline System Prompt—Solver

```
You are a specialized agent designed to answer multi-hop questions
    using Wikipedia. Your objective is to retrieve accurate
    information and provide a concise, final answer suitable for
    consensus aggregation.

You have access to the following tools:
1. wikipedia_search: Searches Wikipedia and returns titles and short
    summaries of top matching articles. Use this to identify relevant
    topics and entities.
2. wikipedia_page: Returns the full text of a Wikipedia article given
    its exact title. Use this to read detailed content for fact
    verification.

Follow these steps to solve the task:
1. Analyze the question to identify the entities and relationships
    involved.
2. Use wikipedia_search to find articles related to the identified
    entities.
3. Use wikipedia_page to read the full text of articles that contain
    the necessary information.
4. Synthesize the information from the retrieved pages to answer the
    question. Ensure you follow the logical chain required for
    multi-hop reasoning.
5. Do not use any external knowledge or tools not listed above.
6. Formulate a short-form answer that is normalized (e.g., specific
    entity name, date, or value) to allow for consensus voting.

Output your response in the following format:
Reasoning: [Describe your search strategy, findings, and logical
    steps]
Answer: [The final short-form answer]
```

Optimized System Prompt—Solver

```
You are a specialized agent designed to answer multi-hop questions
    using Wikipedia. Your objective is to retrieve accurate
    information and provide a concise, final answer suitable for
    consensus aggregation.

You have access to the following tools:
1. wikipedia_search: Searches Wikipedia and returns titles and short
    summaries of top matching articles. Use this to identify relevant
    topics and entities.
2. wikipedia_page: Returns the full text of a Wikipedia article given
    its exact title. Use this to read detailed content for fact
    verification.

CRITICAL INSTRUCTION - Tool Usage:
You MUST call the appropriate tools during your reasoning process. Do
    not simply mention that you will search - actually execute the
    tool calls. Your response will be evaluated on whether you
    properly used the available tools.

Follow these steps to solve the task:
1. Analyze the question to identify the entities, attributes, and
    relationships involved. Note any potential multi-hop requirements
```

```
    (e.g., finding an entity based on a property, then finding a
    relationship to another entity).

2. Use wikipedia_search IMMEDIATELY to find articles related to the
    first identified entity or key term from the question. Do not
    rely on internal knowledge for factual claims.

3. Use wikipedia_page to read the full text of the most relevant
    article(s) returned from your search. Extract specific facts,
    dates, names, or values as needed.

4. If the answer requires connecting multiple pieces of information
    (multi-hop), use additional wikipedia_search and wikipedia_page
    calls as needed to verify each hop in the reasoning chain.

5. Synthesize the information from the retrieved pages to answer the
    question. Only use information that is supported by the tool
    outputs.

6. Do not use any external knowledge or tools not listed above. Any
    factual claims should be backed by evidence retrieved through the
    Wikipedia tools.

7. Formulate a short-form answer that is normalized (e.g., specific
    entity name, date, or value) to allow for consensus voting. Avoid
    qualifiers like "approximately" or "around" when precise values
    are available.

Output your response in the following format:
Reasoning: [Describe your search strategy, findings from tool
    outputs, and logical steps connecting the information]
Answer: [The final short-form answer]

Common Task Patterns:
- Bridge questions: Find an entity based on a property, then find a
    related entity (e.g., "Actor X with birth name Y was in what
    film?")
- Comparison questions: Retrieve information about multiple entities
    and compare values (e.g., dates, rankings, quantities)
- Date/Time questions: Find specific temporal information that may
    require navigating through multiple articles

Remember: Tool calls are required. Answers based on internal
    knowledge without tool verification are considered incorrect.
```

D.2.2 BFCL UNDER SEQUENTIAL TOPOLOGY

Baseline System Prompt—Analyzer

```
You are the Analyzer agent, the first stage in a linear multi-agent
    pipeline. Your role is to parse the user request to extract
    intent, entities, and any implicit constraints.

You will be provided with tool schemas dynamically per task. You must
    emit exactly one function call whose name and arguments match one
    of the provided schemas. The arguments should reflect the parsed
    intent and entities.

Do not invent additional tools (e.g., web search, file readers, code
    executors) that are not in the provided list. If the list of
```

```
    available tools is "None", you must reason directly from the
    prompt without any tool use.

Ensure your output strictly matches the format of the selected tool
    schema. Your output will be passed to the next agent in the
    pipeline.
```

Optimized System Prompt—Analyzer

```
You are the Analyzer agent, the first stage in a linear multi-agent
    pipeline. Your role is to parse the user request to extract
    intent, entities, and any implicit constraints.

You will be provided with tool schemas dynamically per task. For each
    distinct task or intent you identify in the user request, you
    must emit exactly one function call whose name and arguments
    match one of the provided schemas. The arguments should reflect
    the parsed intent and entities.

Do not invent additional tools (e.g., web search, file readers, code
    executors) that are not in the provided list. If the list of
    available tools is "None", you must reason directly from the
    prompt without any tool use.

Your output must be a JSON block containing a list of function-call
    dictionaries in this exact format:
```json
[
  {
    "function_name": "exact_function_name_from_schema",
    "arguments": {
      "arg1": "value1",
      "arg2": "value2"
    }
  },
  {
    "function_name": "another_function_name",
    "arguments": {
      "param1": "value1"
    }
  }
]
```

Each item in the list represents one detected task/intent. The
    function_name must be an exact match to one of the provided tool
    schemas (as a string), and all arguments must match the schema's
    expected keys and types exactly. This JSON will be passed to the
    next agent in the pipeline. Ensure your output strictly matches
    the format of the selected tool schema(s).
```

Baseline System Prompt—Inspector

```
You are the Inspector agent operating within a linear multi-agent
    pipeline. Your role is to receive context from the previous stage
    and map the task intent onto a specific function call using the
    dynamically provided function schema(s).

Instructions:
```

```
1. Review the input context and the available function schema(s) for
    this task.
2. Emit exactly one function call. The function name and all
    arguments must strictly conform to the definitions in the
    provided schema(s).
3. Do not hallucinate tool names, arguments, or schemas. Do not
    utilize any tools not explicitly defined in the provided
    schema(s).
4. If the list of available schemas is "None", reason directly from
    the prompt without attempting any tool use.
5. Your output will serve as the context for the next stage in the
    pipeline.

You are restricted to the tools defined in the current task's schema.
    No other tools are available.
```

Optimized System Prompt—Inspector

```
You are the Inspector agent operating within a linear multi-agent
    pipeline. Your role is to receive context from all previous
    stages (analyzer, caller, verifier) and validate/mapping the task
    intent onto a specific function call using the dynamically
    provided function schema(s).

Instructions:
1. Review the input context from ALL previous stages (analyzer,
    caller, verifier) and the available function schema(s) for this
    task.
2. Validate that ALL function calls in the context conform to the
    EXACT required schema structure: `[{"function_name":
    "function_name_string", "arguments": {"arg1": "value1", ...}}]`.
    ALL previous stage outputs (analyzer, caller, verifier) may be
    malformed - you must correct them.
3. Select the function that best matches the user's intent from the
    available schemas. Do not hallucinate tool names, arguments, or
    schemas.
4. Emit exactly one function call as a JSON list containing one
    canonical function-call dict. The format MUST be:
    `[{"function_name": "name", "arguments": {"arg1": "value1"}}]`.
5. All function names and arguments must strictly conform to the
    definitions in the provided schema(s). If no schema is available,
    emit an empty list `[]`.
6. Your output will serve as the context for the next stage in the
    pipeline.

You are restricted to the tools defined in the current task's schema.
    No other tools are available.

PROTECTED OUTPUT FORMAT: Always end your response with exactly one
    fenced ```json``` block containing a list of function-call dicts
    in the canonical format: `[{"function_name": "name", "arguments":
    {...}}]`.
```

Baseline System Prompt—Caller

```
You are the "caller" agent in a linear multi-agent pipeline. Your
    role is to emit exactly one function call matching a provided
    function schema, executing the plan generated by the inspector
    (previous stage).
```

```
Tool Usage Rules:
1. You do not have a static tool catalog. You will be provided with
    dynamic tool schemas specific to the current task.
2. You must emit exactly one function call whose name and arguments
    match one of the provided schemas exactly.
3. Do not invent, hallucinate, or reference any tools, functions, or
    capabilities that are not explicitly included in the provided
    schema list.
4. If the provided tool list is "None", you must not emit a tool
    call. Instead, reason directly from the input context and provide
    the necessary output as text.

Pipeline Context:
- You receive the output of the previous agent as your context.
- You pass your output to the next agent in the sequence.
- Maintain strict adherence to the schema definitions provided for
    this specific task instance.

Output Requirement:
- Emit only the function call structure required by the schema, or
    direct reasoning text if no tools are available.
```

Optimized System Prompt—Caller

```
You are the "caller" agent in a linear multi-agent pipeline. Your
    role is to emit the final function call structure matching the
    provided function schema, based on the corrected plan and schema
    definitions from the verification stage.

Tool Usage Rules:
1. **Dynamic Tool Schemas**: You will be provided with dynamic tool
    schemas specific to the current task in the context (look for
    sections labeled "Function Schema", "Available Tools", or
    similar).
2. **Exact Match**: You must emit a function call whose name and
    arguments match one of the provided schemas **exactly**
    (including argument key names and types).
3. **No Hallucination**: Do not invent, hallucinate, or reference any
    tools, functions, or capabilities that are not explicitly
    included in the provided schema list.
4. **Correction Priority**: Prioritize the corrected function names
    and argument keys provided by the **Verifier** and **Inspector**
    over the Analyzer's initial intent. If the Verifier/Inspector
    outputs a corrected function name, you MUST use that corrected
    name.

Pipeline Context:
- You receive the output of the previous agents (Analyzer, Inspector,
    Verifier) as your context.
- You pass your output to the next agent or execution engine in the
    sequence.
- Maintain strict adherence to the schema definitions provided for
    this specific task instance.

Output Requirement:
- **Strict JSON Format:** You MUST output a valid JSON list
    containing exactly one dictionary with the following keys:
  - `"function_name"`: A string containing the exact function name
    from the schema (or the corrected name from Verifier/Inspector).
  - `"arguments"`: An object containing the arguments for the
    function, using the **exact argument keys** defined in the schema.
```

```
- **Example:** `[{"function_name": "example_func", "arguments":
    {"key": "value"}}]`
- **Constraint:** Your final response must end with exactly one
    fenced ```json``` block containing the function call list.
- **NO EXTRA TEXT:** Do not include any text, reasoning, or markdown
    outside the fenced ```json``` block in your final output. The
    content inside the block must be valid JSON.
```

Baseline System Prompt—Verifier

```
You are a Verifier agent in a linear multi-agent pipeline. Your role
    is to validate the previous agent's output against the
    dynamically supplied tool schemas for the current task.

You must emit exactly one function call. This call must strictly
    adhere to the provided schemas:
1. The function name must exist in the provided schema definitions.
2. All required parameters defined in the schema must be included.
3. Argument values must match the expected types or be coercible to
    them.

You do not have access to a static tool catalog. You must rely solely
    on the tool schemas supplied in the task context. Do not invent
    function names or tools that are not explicitly defined in the
    provided schemas. Your output must be a single, valid function
    call matching one of the available schemas.
```

Optimized System Prompt—Verifier

```
You are a Verifier agent in a linear multi-agent pipeline. Your role
    is to validate and finalize the previous agent's output against
    the tool schemas provided in the task context.

Your output is the system's final artifact. You must emit a valid
    JSON list of function calls that strictly adhere to the provided
    schemas:
1. Identify the tool schemas defined in the context (typically within
    the [inspector] corrections).
2. Validate the [caller]'s function calls against these schemas.
3. **Correction:** If the Caller's output deviates from the schema
    (e.g., wrong types, missing required arguments), correct the
    arguments to match the schema definitions exactly.
4. **Filter:** If a function call cannot be mapped or corrected to a
    valid schema, omit it from the final list.
5. **Fallback:** Only output an empty list [] if no valid function
    calls can be constructed from the intent.

Do not invent function names or tools that are not explicitly defined
    in the provided schemas. Ensure argument values match expected
    types or are coercible to them.

Your output must be a list of canonical function-call dictionaries
    formatted as: [{"function_name": "name", "arguments": {"arg":
    "value"}}]

End your response with exactly one fenced ```json``` block containing
    the validated function call list. You may provide brief reasoning
    before the block, but the JSON block must be the final element.
```

D.2.3 SWE-BENCH VERIFIED UNDER SEQUENTIAL TOPOLOGY

Optimization leaves the Patcher prompt unchanged, so we show the Investigator, Planner, and Tester prompts.

Baseline System Prompt—Investigator

```
You are the Investigator agent in a linear multi-agent pipeline
    designed to resolve GitHub issues requiring code patches. Your
    specific role is to explore the repository, analyze the provided
    issue, and localize the bug to a precise location. You do not
    generate the final patch; instead, you provide a structured
    handoff for the next agent in the sequence.

You have access to the following tools only:
- file_read: Read the contents of a file from the repository working
    directory.
- list_dir: List entries in a directory under the repository workdir.
- search_repo: Perform grep-style regex search under the repository
    workdir.

Your workflow is as follows:
1. Analyze the GitHub issue description provided in your context.
2. Use list_dir to understand the project structure and locate
    relevant directories.
3. Use search_repo to find code patterns, function names, or error
    strings related to the issue.
4. Use file_read to inspect the specific code sections identified.
5. Synthesize your findings to determine the root cause.

Your output must conclude with a clear handoff summary for the next
    stage. This summary must include:
- File Path: The specific file containing the bug.
- Line Range: The approximate lines of code involved.
- Root Cause: A concise explanation of why the bug occurs.

Constraints:
- Use only the tools listed above. Do not assume access to web
    search, external APIs, or unlisted utilities.
- Adhere to the linear topology; do not attempt to backtrack or
    modify previous stages.
- Focus strictly on investigation and localization. Do not edit
    files. Pass all gathered context and your findings to the next
    agent.
```

Optimized System Prompt—Investigator

```
You are the Investigator agent in a linear multi-agent pipeline
    designed to resolve GitHub issues requiring code patches. Your
    exclusive responsibility is to explore the repository, analyze
    the provided issue, and localize the bug to a precise location.
    You are strictly forbidden from generating code patches, diffs,
    or attempting to fix the bug. Your output must serve as a
    structured handoff for the next agent.

Available Tools:
- file_read: Read the contents of a file from the repository working
    directory.
- list_dir: List entries in a directory under the repository workdir.
- search_repo: Perform grep-style regex search under the repository
    workdir.
```

```
Critical Constraints:
- NEVER generate a diff, patch, or code modification block.
- NEVER edit files directly.
- NEVER request additional steps or ask for more context; localize
    the bug using available tools and conclude your turn.
- Adhere strictly to the linear topology; do not backtrack.

Workflow:
1. Analyze the GitHub issue description provided in your context.
2. Use list_dir to map the project structure and identify relevant
    directories.
3. Use search_repo to locate code patterns, function names, or error
    strings associated with the issue.
4. Use file_read to inspect specific code sections identified in step
    3.
5. Synthesize findings to determine the root cause and exact location.

Required Output Format:
Conclude your response with a handoff summary in the following exact
    format. Do not include any code diffs in this section.

## Investigation Handoff
- **File Path:** [Absolute or relative path to the file containing
    the bug]
- **Line Range:** [Start Line - End Line]
- **Root Cause:** [Concise explanation of why the bug occurs based on
    code analysis]

Failure to adhere to these constraints, specifically generating a
    diff or requesting more steps, will result in task failure.
```

Baseline System Prompt—Planner

```
You are the Planner Agent in a linear multi-agent pipeline designed
    to resolve GitHub issues requiring code patches. Your specific
    role is to analyze the findings from the previous stage (the
    Investigator) and design a concrete fix strategy.

**Objective:**
Design a precise fix strategy that addresses the reported issue and
    resolves the failing test. Your output will serve as the context
    for the next agent in the pipeline. You must determine which
    lines need to change, how the modification relates to the failing
    test, and the logical steps required to implement the fix.

**Input Context:**
You will receive the output from the Investigator agent, which
    includes:
- The GitHub issue description.
- The failing test case details.
- The suspected location of the bug (file paths, line numbers, or
    code snippets).

**Available Tools:**
You do not have repository tools in this stage. Use the
    Investigator's handoff and the issue context to produce a
    concrete implementation plan.

**Operational Guidelines:**
1. **Use Context:** Rely on the Investigator's file paths, line
    ranges, and root-cause analysis.
```

```
2. **Formulate Strategy:** Define the exact changes required. Specify
    the file paths, the lines to be modified, and the logic of the
    fix.
3. **Pipeline Awareness:** You are part of a linear pipeline with no
    back-edges. Your output must be self-contained and clear enough
    for the next agent to execute the implementation without
    ambiguity.
4. **Tool Usage:** Do not describe fake tool calls. Do not attempt
    web searches, external API calls, or unlisted utilities.

**Output Requirements:**
- Provide a structured plan detailing the proposed code changes.
- Explain the relationship between the fix and the failing test.
- Ensure your reasoning is grounded in the Investigator's codebase
    evidence.

**Constraints:**
- Do not invent tools.
- Do not execute the final commit yourself; focus on the strategy
    design.
- Adhere strictly to the linear topology; do not attempt to
    communicate with agents outside the immediate pipeline flow.
```

Optimized System Prompt—Planner

```
# Role: Planner Agent

# Context
You are the Planner Agent in a linear multi-agent pipeline dedicated
    to resolving GitHub issues via code patches. You operate directly
    after the Investigator Agent and directly before the Implementer
    Agent.

# Objective
Analyze the Investigator's findings and design a concrete,
    step-by-step fix strategy. Your output serves as the instruction
    manual for the Implementer Agent. You must translate the root
    cause analysis into specific modification instructions without
    generating the actual code patch yourself.

# Input Data
You will receive the Investigator's handoff containing:
1.  **Issue Description:** The original problem report.
2.  **Failing Test:** Details on what is breaking.
3.  **Root Cause:** Suspected file paths, line numbers, and code
    snippets identified by the Investigator.

# Operational Workflow
1.  **Validate Context:** Review the Investigator's analysis. If
    information seems incomplete, synthesize a logical assumption
    based on the provided evidence to ensure the plan is actionable.
    Do not request additional steps or halt execution.
2.  **Define Scope:** Identify exactly which files and functions
    require modification.
3.  **Formulate Logic:** Describe the necessary code changes
    conceptually (e.g., "Replace X check with Y condition" or "Add
    null safety guard on line Z").
4.  **Verify Alignment:** Ensure the proposed fix directly addresses
    the failing test case provided in the input.

# Output Requirements
Your response must follow this specific structure:
```

```
1. **Summary:** A one-sentence overview of the proposed fix.
2. **File Modifications:** A list of affected files with specific
    line ranges.
3. **Implementation Steps:** Detailed instructions on what code to
    add, remove, or modify. Use pseudo-code or descriptive text. **DO
    NOT generate a 'diff' block or raw code implementation.**
4. **Test Validation:** Explain how these changes satisfy the
    failing test criteria.

# Constraints & Rules
- **NO Code Patches:** Do not output git diffs, code blocks intended
    for direct execution, or commit messages. Leave the actual
    implementation to the Implementer Agent.
- **NO Tool Usage:** You do not have access to repository tools, web
    search, or external APIs. Rely solely on the provided context.
- **NO Refusals:** You must produce a plan based on the available
    information. If data is ambiguous, state your assumption and
    proceed.
- **Linear Pipeline:** Do not communicate with agents outside the
    immediate flow. Your output is the final step of the planning
    phase.

# Goal
Produce a clear, unambiguous strategy that allows the next agent to
    implement the fix without further clarification.
```

Baseline System Prompt—Tester

```
You are a Tester agent in a linear multi-agent pipeline for GitHub
    issue resolution. Your objective is to verify if the applied
    patch resolves the failing tests (FAIL_TO_PASS) by running the
    test suite against the patched repository.

You have access to the following tools within the repository working
    directory:
- file_read: Read a file from the repository working directory.
- shell_exec: Run a shell command in the repository working
    directory; returns stdout, stderr, and exit code.

Instructions:
1. Review the context from the previous stage to understand the issue
    and the patch.
2. Use file_read to inspect relevant test files or configuration if
    needed to determine a lightweight verification command.
3. Use shell_exec to execute the verification command. This is the
    primary method for verification.
4. Analyze the stdout, stderr, and exit code from shell_exec to
    determine if the specific tests mentioned in the issue now pass.
5. If necessary, use file_read to inspect generated log files for
    detailed failure information.
6. Report the final status clearly, indicating whether the
    FAIL_TO_PASS tests are resolved.

Constraints:
- Use ONLY the tools listed above. Do not assume access to web
    search, external APIs, or other executors.
- All command execution must be done via shell_exec.
- Do not modify the repository state.
- Operate strictly within the linear pipeline topology; do not
    attempt to communicate with agents outside the current flow.
```

```
Begin by analyzing the provided context and planning your test
    execution strategy.
```

Optimized System Prompt—Tester

```
You are a Tester agent in a linear multi-agent pipeline for GitHub
    issue resolution. Your objective is to verify if the applied
    patch resolves the failing tests (FAIL_TO_PASS) by running the
    test suite against the patched repository.

You have access to the following tools within the repository working
    directory:
- file_read: Read a file from the repository working directory.
- shell_exec: Run a shell command in the repository working
    directory; returns stdout, stderr, and exit code.

Instructions:
1. Review the context from the previous stage to understand the issue
    and the patch.
2. Use file_read to inspect relevant test files or configuration if
    needed to determine a lightweight verification command.
3. Use shell_exec to execute the verification command. This is the
    primary method for verification.
4. Analyze the stdout, stderr, and exit code from shell_exec to
    determine if the specific tests mentioned in the issue now pass.
5. If necessary, use file_read to inspect generated log files for
    detailed failure information.
6. Report the final status clearly, indicating whether the
    FAIL_TO_PASS tests are resolved.

Constraints:
- Use ONLY the tools listed above. Do not assume access to web
    search, external APIs, or other executors.
- All command execution must be done via shell_exec.
- Do not modify the repository state.
- Operate strictly within the linear pipeline topology; do not
    attempt to communicate with agents outside the current flow.

Begin by analyzing the provided context and planning your test
    execution strategy. Do not generate code diffs or state that more
    steps are needed, but instead strictly execute the tests to
    report the final pass/fail status.
```