MAS-ZERO: Designing Multi-Agent Systems with Zero Supervision

Zixuan Ke, Austin Xu, Yifei Ming, Xuan-Phi Nguyen, Caiming Xiong, Shafiq Joty
Salesforce AI Research
Correspondence: zixuan.ke@salesforce.com

Project Page: https://mas-design.github.io/

1,000+ Discovered MAS Designs: https://mas-design.github.io/mas_collection.html

Abstract

Multi-agent systems (MAS) leveraging the impressive capabilities of Large Language Models (LLMs) hold significant potential for tackling complex tasks. However, most current MAS depend on manually designed agent roles and communication protocols. These manual designs often fail to align with the underlying LLMs' strengths and struggle to adapt to novel tasks. Recent automatic MAS approaches attempt to mitigate these limitations but typically necessitate a validation set for tuning and yield static MAS designs lacking adaptability during inference. We introduce MAS-ZERO, the first self-evolved, inference-time framework for automatic MAS design. MAS-ZERO employs meta-level design to iteratively generate, evaluate, and refine MAS configurations tailored to *each problem instance*, without requiring a validation set. Critically, it enables dynamic agent composition and problem decomposition through meta-feedback on solvability and completeness. Experiments across math, graduate-level QA, and coding benchmarks, using both closed-source and open-source LLM backbones of varying sizes, demonstrate that MAS-ZERO outperforms both manual and automatic MAS baselines, achieving a 7.44% average accuracy improvement over the next strongest baseline while maintaining cost-efficiency. These findings underscore the promise of meta-level self-evolution in MAS design.

1 Introduction

While standalone large language models (LLMs) have demonstrated strong performance across numerous tasks [9, 20, 36], many problems remain too intricate for a single model to solve effectively [38, 14]. To tackle these challenges, the exploration of multi-agent systems (MAS) composed of multiple LLM agents has gained increasing traction among researchers [19].[1] These agents often assume distinct *roles*, such as generator or verifier [35], engage in debates offering varied perspectives [30, 37], and perform assigned subtasks [22].

A fundamental challenge in MAS lies in designing an effective connection and configuration of these agents to solve a given problem. Initially, MAS were handcrafted, with humans designing both agent roles and inter-agent communication protocols. However, MAS composed entirely of such manually designed configurations have faced issues such as poor problem specification and

[1] Agents in a MAS can interact with external environmental tools e.g., search tools [17], or collaborate with other agents to address tasks requiring diverse capabilities or multiple steps [23, 5]. This work focuses on the latter scenario, where each agent within the MAS is an LLM communicating with other LLM agents.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Scaling Environments for Agents (SEA).

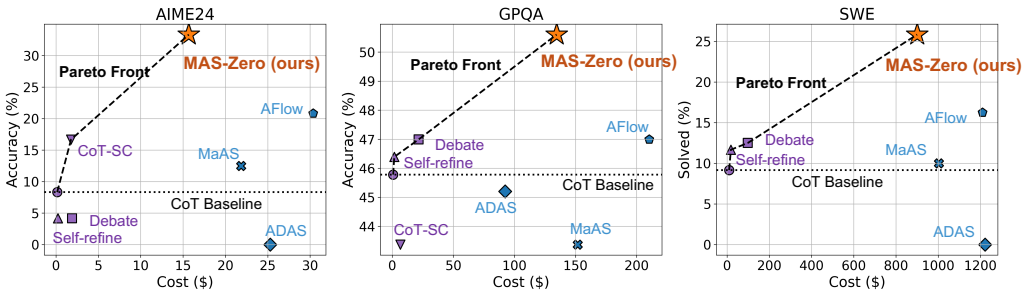

Figure 1: Scatter plots comparing the Pareto fronts of various GPT-4o-based systems on three benchmarks. Manual MAS are marked in purple and automatic MAS in blue. MAS-ZERO is highlighted as an orange star. MAS-ZERO delivers high performance at lower cost than comparable automatic MAS methods, establishing a new frontier for accuracy vs. cost trade-off.

inter-agent misalignment [3, 31], especially when the MAS agents are not specifically trained with such configurations.

These shortcomings are understandable, as manually specifying agent roles, communication protocols, and problem decomposition is difficult when the human designer and the underlying LLMs are not well aligned. Moreover, manual approaches do not scale well to novel problems, especially as the problems become more complex. Recent work has explored automatic MAS design, often framing it as a pruning problem given a fixed and fully connected set of human designed MAS [51, 47, 44] or a generation problem, where the MAS is generated by a central *meta agent* that orchestrates system design [15, 47].

However, these automatic MAS design systems have significant limitations. Most rely on a "training" phase with labeled validation sets to tune configurations, which are often unavailable in real-world scenarios and may not generalize. This training, based solely on *outcome* correctness, provides limited insight into the system's internal dynamics. Furthermore, the training often yields a *fixed* architecture (i.e., one for the entire problem set) which lacks per-problem adaptability at test time.[2] While these limitations might be less evident on simple tasks such as problems in GSM8K [8] and HumanEval [6], commonly used for evaluation [15, 44, 47]. Figure 1 demonstrates that existing systems falter on complex problems requiring multi-step planning and task decomposition. Alarmingly, many methods show little to no improvement over simple CoT with the base LLM, meaning the integrated system does not even outperform a single component.

To overcome these limitations, we argue that an effective MAS should satisfy two core desiderata: **(1)** automatic design of agent structures *aligned with the underlying LLM's capabilities*; and **(2)** adaptivity to the specific problem. In this work, we propose a novel automatic inference-time MAS optimization framework, called MAS-ZERO, which designs MAS with *zero* supervision, while satisfying all the aforementioned desiderata. In particular, MAS-ZERO introduces a meta-agent that iteratively understands the limitations of individual agents and their combinations, and refines the MAS design accordingly. This process operates entirely at test time without relying on a validation set, allowing for unique MAS designs *per-problem* and overcoming the limited adaptivity of prior work. While this inference-time mechanism does incur higher token usage during testing, it avoids expensive validation-time optimization and shifts the design effort to the testing phase, where it can flexibly handle new tasks, and often be more effective [1]. Such a trade-off is inherent to any inference-time approach and has demonstrated strong potential [24]. Crucially, MAS-ZERO maintains a self-evolving process to build a compound agentic system, dynamically modifying its architecture in response to intermediate execution signals, rather than relying on static prompts or pre-defined templates.

To achieve this, MAS-ZERO tasks a **meta-agent** with multiple roles. Figure 2 illustrates a conceptual overview and contrasts MAS-ZERO with both automatic and manual MAS designs. Specifically, MAS-ZERO involves two key steps:

[2] [44] uses sub-networks for per-problem adaptivity, but the core MAS structure remains fixed.

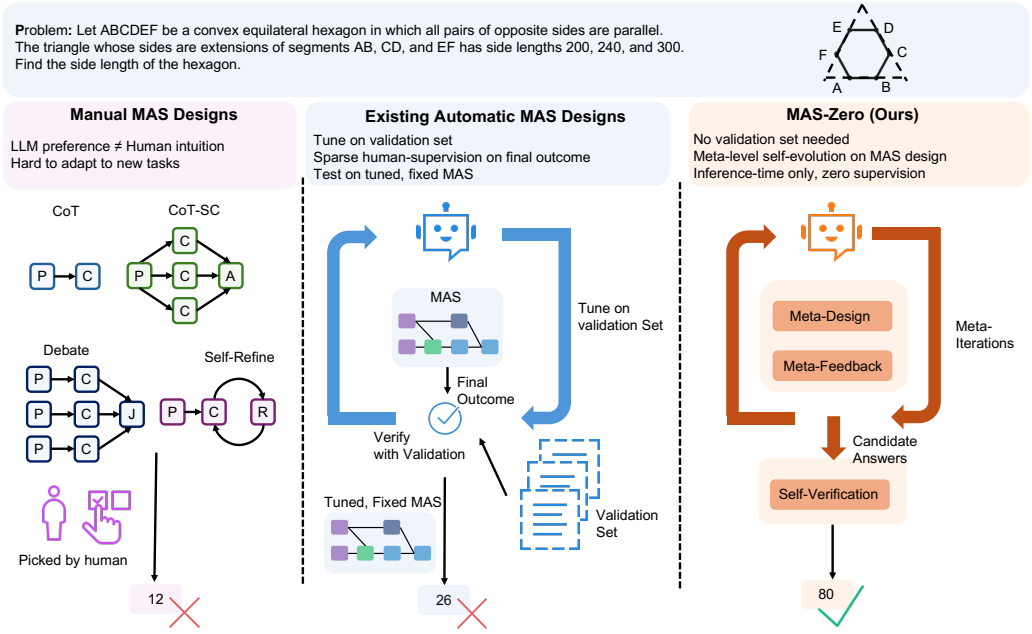

Figure 2: Conceptual comparison of MAS-ZERO, with existing automatic and manual MAS designs. In manual MAS, "P" indicates the problem. "C" denotes the CoT generator, "A" denotes the majority aggregator in CoT-SC, "J" denotes the final judge in a debate system, and "R" denotes the reflector in the self-refine system. MAS-ZERO avoids tuning MAS on validation set by maintaining a self-evolving process that iteratively designs and evaluates task-specific MAS at inference time.

- **Meta-iterations**: The meta-agent iteratively conducts two phases: **(a) Meta-design**: **Decomposition** of the given task and **generation** of a corresponding MAS implementation (as executable code), allowing the system to adapt dynamically to each new task. **(b) Meta-feedback**: **Evaluating** the generated MAS design for informative meta-level feedback, assessing two key criteria: **solvability** (can the current MAS solve the problem?) and **completeness** (does the current MAS cover all necessary steps?). Each iteration of this process produces a candidate solution to the task.
- **Self-verification**: The meta-agent selects the best outcome from the set of all candidate solutions obtained throughout the meta-iterations.

Evaluations across three datasets covering diverse domains (math, graduate-level QA, and code) and multiple LLM backbones (including both closed- and open-source models of varying sizes, such as 32B, 70B, and GPT-4o) show that our model outperforms manual and other automatic MAS baselines by a minimum of 7.44% on average. Furthermore, MAS-ZERO consistently lies on the Pareto frontier of accuracy and cost (Figure 1), effectively gaining a deeper understanding of both strengths and weaknesses of the current MAS design over iterations to enhance future designs.

MAS-ZERO also provides insights into MAS design. For example, we observe the "*MAS moment*", where MAS-ZERO learns to decompose a new question and assign appropriate sub-MAS modules to each sub-question dynamically (see Fig. 4). This type of dynamic assignment would have been difficult to design manually. We also find that off-the-shelf LLMs, struggle to design effective MAS when used directly, but with MAS-ZERO, the MAS performance improves significantly; Additionally, by utilizing a verifier instead of validation set tuning, MAS-ZERO is poised to leverage future improvements in verification. Despite achieving state-of-the-art (SoTA) with a relatively weak self-verifier, our analysis indicates significant potential for further gains as stronger verifiers become available. In summary, our key contributions are:

- We introduce MAS-ZERO, to our knowledge, the **first inference-time-only** automatic MAS design framework. It works in a fully self-evolved way by learning from the behavior of the underlying LLM agents *at inference-time*, enabling per-instance adaptivity with *zero* supervision.

- We present **a new SoTA automatic MAS system** that achieves substantial performance gains over both manually designed and strong automatic baselines, while remaining cost-efficient and Pareto-optimal across LLMs and domains.
- We evaluate MAS-ZERO on various domains and LLMs, presenting **key insights** from the self-evolving process, including the MAS moment, gains from meta-iterations, upper bounds with an oracle verifier, and generalizability across model strengths and agentic setups.

2 Related Work

Manual MAS design. Building on the success of single-agent systems (e.g., CoT [40], self-consistency (CoT-SC) [39]), studies have shown that grouping multiple LLM agents into a MAS can substantially improve individual agent performance. To this end, a variety of human-designed MAS approaches have been proposed [41, 48, 27]. Representative work includes LLM debate [10], and self-refine [28]. However, manually designed MAS are often limited in adaptability and scalability. More importantly, human designers typically do not fully understand the capabilities or limitations of the LLM agents, and may apply subjective preferences that do not align with optimal agent behavior. This makes effective MAS design highly challenging, if not impossible, without automation.

Automatic MAS design. We broadly categorize recent work on automatic MAS design into two families: *pruning-* and *generation-based* approaches.[3]

Pruning-based. This family starts with a fully connected, pre-defined graph where nodes represent LLM agents or human-designed blocks (e.g., debate), and edges represent information flow. The goal is to prune the graph to identify the most effective connections. Recent work includes MASS [49] uses rejection sampling based on validation set performance, while MaAS [44] extends MASS with a question-wise masking mechanism, allowing different questions to activate different subnetworks. While these methods are typically easier to train, they are heavily constrained by the pre-defined structure, which is suboptimal for many tasks.

Generation-based. In this family, a meta-agent LLM generates MAS from scratch, offering greater flexibility in defining novel agents and architectures compared to pre-defined structures. However, this expanded design space presents significant learning challenges. Recent efforts including ADAS [15] and AFlow [47] frames MAS generation as a code generation task. ADAS stores and searches historical designs based on validation performance, while AFlow enhances this with Monte Carlo Tree Search. Our framework also adopts a code-based representation for MAS design. Unlike prior work that uses a potentially unreliable validation set, MAS-ZERO uses self-evolving approach to infer agent capabilities for meta-level design at inference-time. In addition, MAS-ZERO incorporates question decomposition into MAS design, allowing it to construct and refine MAS at the sub-task level, which is not supported by existing automatic MAS systems.

3 MAS-ZERO Framework

As shown in Fig. 3, MAS-ZERO takes a question and a seed set of MAS *building blocks* (i.e., established human-designed blocks like CoT-SC to define the design space) as inputs, ultimately producing the final answer. These inputs are processed by the central **meta-agent**, which orchestrates both the **meta-iteration** (Sec. 3.1) and **self-verification** (Sec. 3.2) steps. Importantly, the whole process is functioned *without* prior knowledge or internal details of the underlying LLM agents.

3.1 Meta-Iterations Step

Initially lacking knowledge of the underlying LLM agents' internal capabilities or limitations, the meta-agent learns the component agents' potential by observing both sub-task level and *finer* agent-level performance. MAS-ZERO achieves this through an iterative process. Each iteration comprises two main phases: (1) **meta-design** (Sec. 3.1.1), where the meta-agent decomposes the given question into sub-tasks and proposes a MAS, based on the building blocks and any available feedback from prior iterations, to address each sub-task; and (2) **meta-feedback** (Sec. 3.1.2), where the meta-agent reviews the proposed MAS and sub-tasks to generate feedback. Such feedback is

[3] We omit methods that involve updating LLM parameters for MAS design, e.g., [42, 12], since MAS-ZERO does not require any training. Early methods like DyLAN [25] are discussed in App. B.

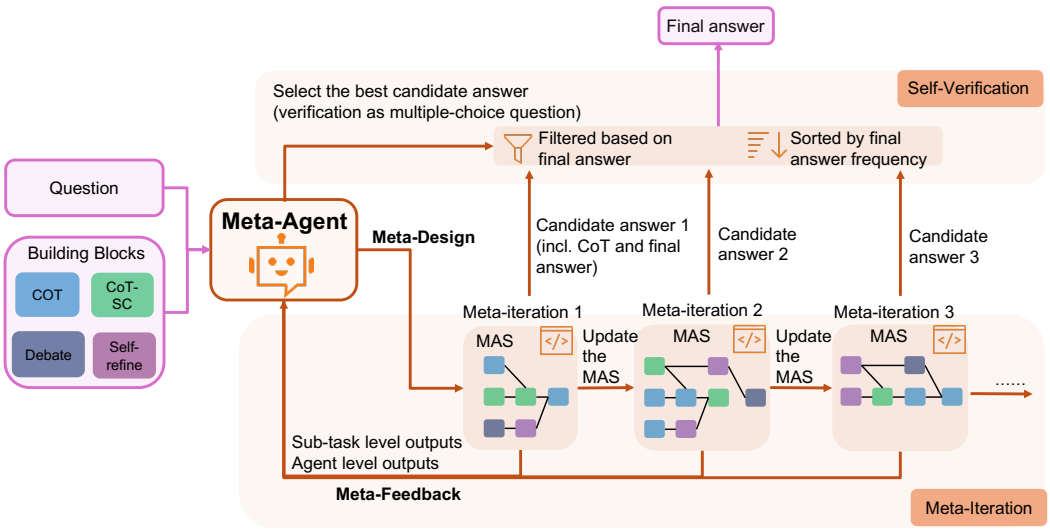

Figure 3: MAS-ZERO overview. MAS-ZERO takes as inputs the question and building blocks as the seed set of MAS, and solves the task by self-evolving. It operates in two key steps: **(1) Meta-iteration**, where the meta-agent designs sub-tasks and their corresponding sub-MAS, while also providing meta-level feedback on solvability and completeness. **(2) Self-verification**, where the meta-agent verifies the final outcome from a set of candidate answers.

based on intermediate outputs and reveals how the current MAS performs across various sub-task decompositions and configurations, acting as a valuable signal to guide future iterations. Through the iterative cycle of meta-design and meta-feedback, the MAS is progressively refined to better solve the question. In each round, the meta-agent can adjust the decomposition or MAS configurations (e.g., architecture, LLM agent temperature) based on the feedback received (an example is given in Fig. 4).

3.1.1 Meta-Design Phase

Task decomposition and sub-MAS assignment. Unlike existing work that tackles complex problems all at once, MAS-ZERO explicitly decomposes the original question into *manageable* yet *interdependent* sub-tasks, and generating or assigning one (or more) agents, referred to as a *sub-MAS*, to solve each sub-task. During this phase, the meta-agent not only decomposes questions, but specifies necessary coordination logic between agents, ensuring *specialized* yet *aligned* agents. While the quality sub-tasks and sub-MAS inherently depends on the capabilities of the underlying LLM-powered meta-agent, MAS-ZERO is designed to iteratively evaluate and refine both the decomposition and corresponding sub-MAS designs, which can correct initially imperfect designs, as shown in Fig. 4.

Constraining the search space with building blocks. MAS-ZERO is an iterative approach that improves upon itself. As such, it is important for designs to not vary extremely between iterations. To achieve this, MAS-ZERO limits changes to either updating sub-task or updating the sub-MAS w.r.t. the given *building blocks* (i.e., CoT, CoT-SC, debate, and self-refine in this work).[4] After performing task decomposition, the meta-agent is restricted to modifying connections between different building blocks or adjusting building blocks' parameters (e.g., temperature, number of debate rounds, etc.). This deliberate constraint, informed by our preliminary experiments, balances exploration with improvement: The meta-agent should not invent "new" agent roles that closely resemble existing ones, nor should it simply pruning a fixed architecture. Rather, it is given freedom to analyze and decompose questions then assign new seed-informed sub-MAS to solve sub-tasks.

[4]MAS-ZERO uses code structures as representations for each agent. Generating correct MAS code can be challenging. We provide a code template and utility functions so that the meta-agent only needs to fill in a specific `forward` function. We also perform sanity checks, including syntax validation and field consistency. More details can be found in App. H.

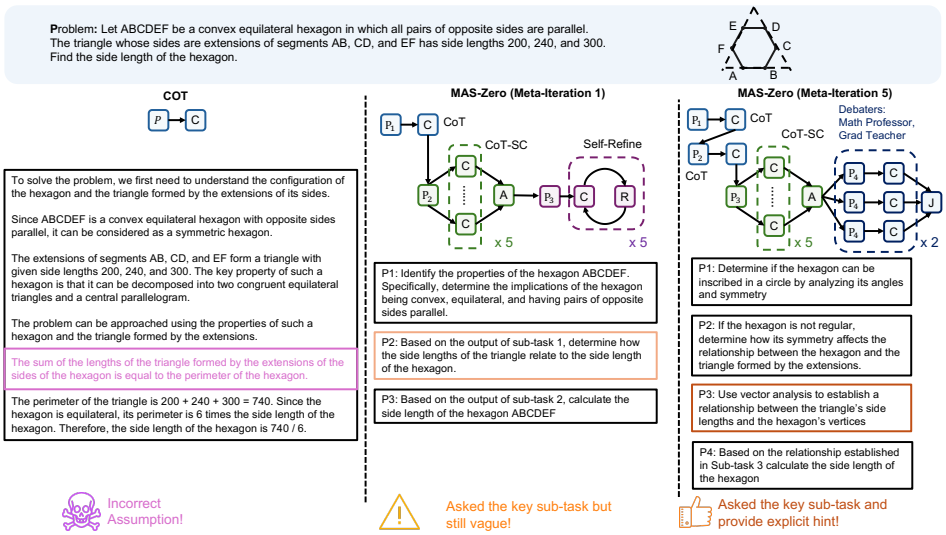

Figure 4: A "MAS moment". MAS-ZERO learns to decompose the task into 4 sub-tasks and dynamically assign appropriate sub-MAS: CoT for the first two, CoT-SC (sampling 5 completions) for the third, and Debate (2 rounds with a math professor and a graduate teacher as debaters) for the fourth.

Updating the MAS design. Importantly, the meta-design is not a prompting heuristic. Instead, the meta-agent dynamically adapts its decomposition and sub-MAS assignments over iterations: After the first meta-design iteration, meta-feedback is collected (detailed in Sec. 3.1.2). In the next iteration, we perform meta-design again, with meta-feedback as additional context to facilitate self-evolution.

3.1.2 Meta-Feedback Phase

A key challenge in MAS design is that the meta-agent only has access down to the agent-level: It does not have access to an agent's internal memory or knowledge. Unlike conventional evaluation strategies that rely only on final outcome signals, in MAS-ZERO, we propose to use *sub-task level* and *agent-level* signals, i.e., the inputs and outputs to sub-MAS and agents, as *proxies* for agent capabilities and limitations. By determining how effective the current MAS is by scrutinizing interactions between agents and sub-MAS components, the meta-agent obtains a richer look into both strengths and weaknesses in the overall MAS. To facilitate this, the meta-agent must first obtain *intermediate MAS outputs*, from which it can perform this fine-grained, intermediate evaluation. MAS-ZERO centers two key criteria: *solvability* (whether the MAS is capable of solving these sub-tasks) and *completeness* (whether the generated sub-tasks are sufficient to reach the final answer to the original question). Targeted feedback from these criteria is then used to guide the next iteration of meta-design.

Obtaining intermediate outputs. This involves solving each sub-task by *executing* each sub-MAS, which itself is comprised of one or more agents, producing a *candidate answer* to the original question. This also grants the meta-agent access to two levels of intermediate outputs: One at the *sub-task (sub-MAS)* level, and another at the *agent* level.

Solvability and completeness as meta-reward. Given the above sub-task and agent level outputs, MAS-ZERO evaluates *solvability* and *completeness*. The meta-agent is given agency in determining each metric, which serve as valuable *meta-rewards* that to guide the MAS refinement.

- **Solvability** requires that each sub-task be *independently* and *completely* solvable by its sub-MAS, ensuring that every sub-task yields reliable outputs.[5] The ability in recognizing a sub-task is too difficult ultimately depends on the ability of the underlying LLMs to recognize and abstain, which

[5]Since MAS-ZERO operates in a self-evolved setting, the meta-agent itself is responsible for assessing solvability. To aid the meta-agent, we allow each agent to output a special token, `[TOO HARD]`, if it determines that the assigned sub-task is beyond its current capabilities.

LLMs	GPT-4o				Llama3.3-70B				Qwen2.5-32B		
Methods	AIME24	GPQA	SWE	**Avg.**	AIME24	GPQA	SWE	**Avg.**	AIME24	GPQA	**Avg.**
CoT	8.33	45.78	9.17	23.26 (↑12.55)	16.67	50.60	2.92	22.09 (↑9.58)	12.50	50.00	45.26 (↑3.69)
CoT-SC	16.67	43.37	—	—	29.17	51.20	—	—	16.67	49.40	45.27 (↑3.68)
Debate	4.17	46.99	12.50	25.35 (↑10.46)	20.83	50.60	6.67	24.42 (↑7.25)	8.33	49.40	44.21 (↑4.74)
Self-Refine	4.17	46.39	11.67	24.65 (↑11.16)	29.17	**54.22**	1.67	23.49 (↑8.18)	16.67	50.60	46.31 (↑2.64)
ReConcile	12.50	48.43	—	—	33.33	47.17	—	—	12.50	47.17	42.79 (↑6.16)
MaAS	12.50	43.37	10.00	23.02 (↑12.79)	33.33	43.98	5.00	21.63 (↑10.04)	20.83	46.99	43.68 (↑5.27)
ADAS	×	45.20	×	×	8.30	53.60	×	×	12.50	47.00	42.64 (↑6.31)
AFlow	20.83	46.99	16.25	28.37 (↑7.44)	33.33	47.59	6.67	23.95 (↑7.72)	**33.33**	48.80	46.84 (↑2.11)
MAS-ZERO	**33.33**	**50.60**	**25.83**	**35.81**	**37.50**	52.41	**16.74**	**31.67**	29.17	**51.81**	**48.95**

Table 1: Overall results. "Avg." denotes the *weighted average*, where weights are based on the number of samples across benchmarks. "—" denotes non-applicable (e.g., CoT-SC is not applicable to SWE). "×" indicates ADAS achieved 0% accuracy, despite being tuned on the validation set. "↑" indicates the difference (improvement) that MAS-ZERO achieves compared to the baselines. Highlighting indicates manual MAS design, automatic MAS design, and our method. SWE is not included for Qwen2.5-32B due to its small maximum context length (32K). Standard deviations on AIME24 are reported in App. F.

may be imperfect. However, we show via ablations in Sec. 4.2 that such signals serve as reasonable proxies for solvability and contribute meaningfully to overall performance.

- **Completeness** requires that the complete set of sub-tasks covers all necessary information from the original input, ensuring that their answers can produce a correct and comprehensive aggregated answer to the original task. While an individual sub-task may address only part of the necessary content, all critical information must be processed and used at some point in the MAS.

Generating feedback. Based on the meta-rewards of solvability and completeness, the meta-agent generates targeted feedback for specific aspects of the MAS that may require revision: If a sub-task is identified as not solvable, then during the next meta-design iteration, it can either further decompose the sub-task or update the corresponding sub-MAS; solvable sub-tasks and corresponding sub-MAS are left untouched. If the meta-agent finds that the union of sub-tasks misses necessary information, it can refine how the original problem is decomposed to incorporate any identified missing information. Overall, this feedback serves a guide for subsequent meta-design iterations, allowing the overall system to iteratively converge toward an effective decomposition and MAS. Crucially, this entire process occurs without any evaluation of the final answer produced when executing the MAS, relying only on analysis of intermediate outputs.

3.2 Self-Verification Step

At each meta-iteration, the MAS is executed to obtain intermediate outputs and a *candidate answer* (including CoT and final answer). After multiple rounds, MAS-ZERO must determine which candidate answer is the most reliable and complete. Relying on the last iteration (or any single iteration) is suboptimal due to stochastic LLM outputs and ongoing MAS refinement. Instead, MAS-ZERO formulates verification as a *selection* problem and tasks the meta-agent with selecting the most coherent and correct output from the set of candidate answers, which is often more tractable than independently scoring each output [13, 50], especially for challenging questions where correctness is hard to assess in isolation. Specifically, MAS-ZERO first *ranks* candidates by their final answer frequency. This acts as a prior favoring majority responses, a strategy shown to be effective in prior work [39]. It then *filters* out clearly invalid answers (e.g., not among the given options). Finally, it *selects* the best answer from the remaining candidates.

4 Experiments

Setup. We consider both the closed-source **GPT-4o** [29] and the open-source LLMs of various sizes, **Llama3.3-70B** [26] and **Qwen2.5-32B** [32]. To fairly evaluate how well MAS-ZERO performs relative to the underlying LLM used to construct the MAS system, we always use the *same* LLM for both the meta-agent and the individual LLM agents in the MAS.

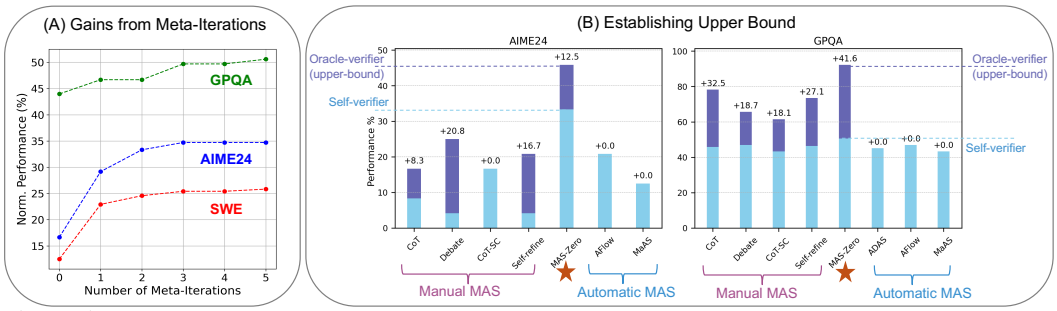

Figure 5: **(A)** Performance gains over meta-iterations. **(B)** Performance gains (purple) over base performance (light blue) given an oracle verifier. Automatic MAS baselines cannot integrate external verifiers, yielding zero improvement.

Benchmarks. We consider three benchmarks across various domains: **AIME24** [2] (math), GPQA-diamond **(GPQA)** [33] (graduate-level QA), and SWE-Bench-Lite-Oracle **(SWE)** [18] (code).[6] To fairly compare with baselines that rely on validation sets, we split each benchmark's original test set into 20% for validation and 80% for testing. For other baselines and MAS-ZERO, which do not use validation sets, we evaluate on the same 80% split.

Baselines. We include four widely used manual MAS baselines: **CoT** [40], self-consistency **(CoT-SC)** [39], **debate** [10] and **self-refine** [28], which also serve as the building blocks in MAS-ZERO, allowing us to clearly observe how our system improves upon the initial configurations. We also include **ReConcile** [5], a popular manual MAS. For automatic MAS design, we include SoTA methods: *pruning-based* **MaAS** [44] and *generation-based* **ADAS** [15] and **AFlow** [47]. We focus on these as they have been shown to outperform other automatic MAS significantly.[7]

4.1 Overall Results

Performance. Table 1 shows the overall results across all LLMs and benchmarks. On average, MAS-ZERO achieves the *best* performance across all LLMs and domains. Below, we summarize the additional takeaways from the comparison:

MAS-ZERO **consistently outperforms all automatic MAS methods.** Across all LLM backbones and benchmarks, MAS-ZERO outperforms SoTA baselines, beating the next best method, AFlow, by 7.44% in on average with GPT-4o as the backbone. The most sizable improvements come **SWE**, with 58% and 149% relative gains over AFlow. The only instance where MAS-ZERO lags is on AIME24 with the Qwen backbone, where it underperforms AFlow by one sample (out of 24 total). Notably, we find that the ADAS is unable to perform certain tasks (0% accuracy), despite being given access to a validation set. Overall, our results highlight the benefit of self-evolving at inference time, a capability absent in prior automatic MAS.

MAS-ZERO **outperforms all manually designed MAS baselines, while others do not.** MAS-ZERO is the only method that *always outperforms CoT* and reliably outperforms other manually designed baselines. One caveat is the self-refine baseline on GPQA with Llama3.3, where correct candidates were generated but ultimately not selected, revealing weaknesses in the Llama backbone's verification ability. AFlow, ADAS, and MaAS alarmingly all underperform the simple CoT baseline on multiple benchmarks, with both ADAS and MaAS *on average* exhibiting degradation from CoT performance. AFlow fares slightly better, but has relatively muted gains when compared to MAS-ZERO. These results suggest that current automatic MAS methods struggle on challenging tasks where MAS-ZERO offers substantive improvements.

Cost-efficiency. Fig. 1 shows the trade-off between performance and cost for GPT-4o across the three benchmarks. Cost is estimated using the latest OpenAI API pricing[8] and includes both "training" (if any) and test-time usage. We observe that MAS-ZERO **lies on the Pareto front across all three datasets**. It is significantly more cost-efficient than AFlow, MaAS, and ADAS, with the lone

[6] Note that self-verification does not apply to SWE, as correctness in SWE is determined directly by the compiler.

[7] Benchmark statistics and more implementation details can be found in App. E.

[8] `https://openai.com/api/pricing/`

Methods	AIME24	GPQA	SWE	Avg.
MAS-ZERO	**33.33**	**50.60**	**25.83**	**35.81**
- decompose	20.83 (↓12.50)	45.18 (↓5.42)	23.75 (↓2.08)	31.86 (↓3.95)
- meta-reward	25.00 (↓8.33)	42.17 (↓8.43)	23.33 (↓2.50)	30.70 (↓5.11)

Table 2: Ablations on components in meta-iteration.

LLM	o3-mini		GPT-4o + Websearch	
Methods	AIME24	GPQA	AIME24	GPQA
CoT	70.00 (↑20.00)	72.22 (↑4.55)	43.33 (↑10.00)	51.52 (↑9.59)
MAS-ZERO	**90.00**	**76.77**	**53.33**	**61.11**

Table 3: MAS-ZERO with stronger agents.

exception of ADAS on GPQA, where the cost increase comes with a 12% accuracy improvement. Of automatic MAS frameworks, MAS-ZERO delivers the highest performance at relatively low cost. While it is expected that automatic MAS methods incur higher costs than manual baselines, MAS-ZERO delivers substantially better performance, making the trade-off highly favorable.

4.2 Ablation Study and Further Analysis

Ablations. To understand the contribution of individual components in MAS-ZERO, we conduct ablations on two key parts of the meta-iteration step (other components and steps are not independently removable): (1) problem decomposition (MAS-ZERO **(-decompose)**) in the meta-design phase. We modify the prompt to ask the meta-agent to propose a MAS configuration without attempting to decompose the question into sub-problems; (2) meta-reward (MAS-ZERO **(-meta-reward)**) in the meta-feedback phase. We alter the prompt so that the meta-agent critiques the current MAS *without* analyzing the solvability and completeness of each sub-task or LLM agent. Table 2 presents the ablation results for GPT-4o. We observe that removing either component leads to a significant drop in performance, indicating that both the problem decomposition and meta-reward mechanisms are critical to the overall effectiveness of MAS-ZERO.

Gains from meta-iterations. We examine whether performance improves over meta-iterations. As shown in Fig. 5 (A) for GPT-4o, performance at iteration 0 is notably lower, suggesting that off-the-shelf LLMs struggle to design effective MAS when used directly. MAS-ZERO progressively improves performance through meta-iterations, demonstrating a strong self-evolving capability at inference time by understanding LLM strengths, decomposing problems, and assigning appropriate sub-MAS.

Establishing upper bound. MAS-ZERO eschews validation-set tuning for self-verification, tasking the meta-agent with selecting a final answer from the set of candidate responses. This allows MAS-ZERO to easily accommodate *external verifiers*. This versatility sets MAS-ZERO up to take advantage of future improvements in verification, unlike existing automatic MAS frameworks, which have no straightforward way of integrating a verifier. Fig. 5 (B) illustrates these potential future gains for GPT-4o. Specifically, when given access to an *oracle verifier* that outputs "correct" or "incorrect" based on the *ground-truth answer*, the performance gap between MAS-ZERO and both manually designed and automatic MAS widens, with GPQA performance approaching 95%.

Stronger Agents. While MAS-ZERO shows strong performance across various LLMs, we are further interested in whether it can improve when using even stronger LLMs as agents. We conduct experiments using a reasoning LLM, **o3-mini** and tool-augmented LLM, **GPT-4o with web-search**. As shown in Table 3, MAS-ZERO consistently outperforms the CoT baselines by a large margin, indicating that the benefits of MAS-ZERO generalize well across model strengths and agentic settings.

5 Conclusion

We presented MAS-ZERO, the first *inference-time-only* automatic MAS design framework with *zero* supervision. Unlike prior work that relies on fixed architectures or validations, MAS-ZERO iteratively refines MAS for each question. Experiments demonstrate its effectiveness and cost-efficiency.

References

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2025.

[2] Team AIME. Aime problems and solutions, 2024.

[3] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent llm systems fail?, 2025.

[4] Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F. Karlsson, Jie Fu, and Yemin Shi. Autoagents: A framework for automatic agent generation, 2024.

[5] 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)*, 2024.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[7] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors, 2023.

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

[9] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[10] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023.

[11] Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Forty-first International Conference on Machine Learning*, 2024.

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

[13] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. A survey on llm-as-a-judge, 2025.

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

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

[16] Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-evolving multi-agent collaboration networks for software development, 2024.

[17] Jinhao Jiang, Jiayi Chen, Junyi Li, Ruiyang Ren, Shijie Wang, Wayne Xin Zhao, Yang Song, and Tao Zhang. Rag-star: Enhancing deliberative reasoning with retrieval augmented verification and refinement. *arXiv preprint arXiv:2412.12881*, 2024.

[18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[19] Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, Caiming Xiong, and Shafiq Joty. A survey of frontiers in llm reasoning: Inference scaling, learning to reason, and agentic systems, 2025.

[20] Zixuan Ke, Yifei Ming, Xuan-Phi Nguyen, Caiming Xiong, and Shafiq Joty. Demystifying domain-adaptive post-training for financial llms. *arXiv preprint arXiv:2501.04961*, 2025.

[21] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023.

[22] Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. Agent-oriented planning in multi-agent systems, 2025.

[23] 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. *arXiv preprint arXiv:2305.19118*, 2023.

[24] Tongxuan Liu, Xingyu Wang, Weizhe Huang, Wenjiang Xu, Yuting Zeng, Lei Jiang, Hailong Yang, and Jing Li. Groupdebate: Enhancing the efficiency of multi-agent debate using group discussion. *arXiv preprint arXiv:2409.14051*, 2024.

[25] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic llm-powered agent network for task-oriented agent collaboration, 2024.

[26] Team Llama. The llama 3 herd of models, 2024.

[27] Cong Lu, Shengran Hu, and Jeff Clune. Intelligent go-explore: Standing on the shoulders of giant foundation models, 2025.

[28] 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, 2024.

[29] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[30] Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Scaling large language model-based multi-agent collaboration, 2025.

[31] Shuofei Qiao, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Benchmarking agentic workflow generation, 2025.

[32] Team Qwen. Qwen2.5 technical report, 2025.

[33] 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 qa benchmark, 2023.

[34] Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. *arXiv preprint arXiv:2410.06153*, 2024.

[35] 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, 2024.

[36] Tu Vu, Kalpesh Krishna, Salaheddin Alzubi, Chris Tar, Manaal Faruqui, and Yun-Hsuan Sung. Foundational autoraters: Taming large language models for better automatic evaluation. *arXiv preprint arXiv:2407.10817*, 2024.

[37] Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities, 2024.

[38] Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. Rethinking the bounds of llm reasoning: Are multi-agent discussions the key?, 2024.

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

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

[41] Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. Expertprompting: Instructing large language models to be distinguished experts, 2025.

[42] Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. Mas-gpt: Training llms to build llm-based multi-agent systems, 2025.

[43] Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. *arXiv preprint arXiv:2406.14228*, 2024.

[44] Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet, 2025.

[45] Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for llm-based multi-agent systems, 2024.

[46] Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks, 2025.

[47] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation, 2025.

[48] Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H. Chi, Quoc V Le, and Denny Zhou. Take a step back: Evoking reasoning via abstraction in large language models, 2024.

[49] Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö. Arık. Multi-agent design: Optimizing agents with better prompts and topologies, 2025.

[50] Yilun Zhou, Austin Xu, Peifeng Wang, Caiming Xiong, and Shafiq Joty. Evaluating judges as evaluators: The jetts benchmark of llm-as-judges as test-time scaling evaluators. *arXiv preprint arXiv:2504.15253*, 2025.

[51] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs, 2024.

A Limitations

While MAS-ZERO demonstrates strong performance and generality across tasks and models, it still has limitations. There are three potential bottlenecks in MAS-ZERO: meta-design, meta-feedback and verification. In meta-design, we showed that using building blocks to constrain the design space can effectively balance exploration and improvement, but it may limit the diversity of emergent MAS designs. In meta-feedback, MAS-ZERO relies on self-evolved and inference-time signals. While iterative refinement helps mitigate this, it may occasionally result in noisy feedback. Finally, in verification, our results show that the current simple self-verifier leaves significant room for improvement toward the oracle upper bound. These aspects present opportunities for future work, such as improving verification mechanisms and introducing meta-agent-specific training to enhance both meta-design and meta-feedback.

B Earlier Related Work

Some prior work treats prompt optimization for individual agents as part of MAS design. Examples include PromptBreeder [11] and DsPy [21]. More recently, this idea has been extended to broader automatic MAS design, where prompt optimization is included either as an additional design step or as part of the search space.

Pruning-based. This line of work has evolved quickly [45, 46, 16]. Earlier examples include GPTSwarm [51] which optimizes graph structures via reinforcement learning but struggles to represent workflows with conditional state due to limitations of static graphs. DyLAN [25] uses message passing to dynamically activate agent compositions, estimating each agent's contribution based on a primary trial, AgentSquare [34] leverages a verifier as a performance predictor to guide the pruning.

Generation-based. Earlier methods like AutoAgents [4] and AgentVerse [7] relied on human-engineered pipelines. EvoAgent [43] later employed evolutionary algorithms to optimize these pipelines.

C MAS-ZERO Algorithm

In Section 3, we provide examples of the prompt and workflow of MAS-ZERO. Algorithm 1 presents the detailed algorithm. Note that we merge meta-feedback and meta-design in meta-iterations into a single `Meta_Feedback_Update` function in the implementation for simplicity.

D Illustration of Meta-Iteration

Figure D.1 gives an illustration. Given a question, the meta-agent is prompted to decompose the task and propose a MAS based on the question and building blocks (see Appendix G for the detailed prompts). The meta-agent then outputs a MAS in the form of code, which can be executed by an external compiler to obtain intermediate and final outputs for the sub-tasks and agents.

After performing meta-decomposition and executing the MAS code via a compiler, both the resulting MAS and its intermediate outputs (sub-task samples and agent samples) are used as input to the meta-feedback phase. We prompted the meta-agent to review the *solvability* and *completeness*. Based on this evaluation, which serves as meta-reward, the meta-agent generates feedback and updates its design in the next iteration.

E Implementations, Benchmarks and Baselines Details

Implementation details. For MAS-ZERO, we first execute each of the four building blocks designs before conducting 5 rounds of iterative refinement. This yields 9 candidate answers, from which our meta-agent selects one final answer. For fair comparison, we sample 9 independent outputs for CoT-SC and take the majority vote. Similarly, both debate and self-refine are run for 9 rounds. All

Algorithm 1: MAS-ZERO: Designing Multi-Agent Systems with Zero Supervision

Input: Question Q, building blockss $\{\mathcal{M}^{(1)}, \ldots, \mathcal{M}^{(k)}\}$, Meta-agent $\mathcal{A}$, Iterations T
Output: Final Answer y^*

Initialize candidate answer list $\mathcal{H} \leftarrow [\,]$;
foreach *building blocks* $\mathcal{M}^{(i)}$ **do**
  $Y_0^{(i)} \leftarrow \texttt{Execute}(\mathcal{M}^{(i)}, Q)$; `// Run each building blocks`
  Append final answer $y_0^{(i)}$ from $Y_0^{(i)}$ to $\mathcal{H}$;
$(\mathcal{Q}_0, \mathcal{M}_0) \leftarrow \mathcal{A}.\texttt{Meta\_Design}(Q, \{\mathcal{M}^{(i)}\})$;
`// Decompose question into sub-tasks and construct initial MAS`
for $t = 1$ **to** T **do**
  $Y_t \leftarrow \texttt{Execute}(\mathcal{M}_{t-1}, \mathcal{Q}_{t-1})$;
  `// Run MAS on sub-tasks`
  Extract sub-task outputs $\{(x_{\text{sub}}, y_{\text{sub}})\}$;
  and agent outputs $\{(x_{\text{agent}}, y_{\text{agent}})\}$ from Y_t;
  $(\mathcal{Q}_t, \mathcal{M}_t, y_t) \leftarrow \mathcal{A}.\texttt{Meta\_Feedback\_Update}($
   $Q, \mathcal{Q}_{t-1}, \mathcal{M}_{t-1},$
   $\{(x_{\text{sub}}, y_{\text{sub}})\}, \{(x_{\text{agent}}, y_{\text{agent}})\},$
   Constraints $= \{\mathcal{M}^{(i)}\})$;
  `// Refine sub-tasks and MAS based on feedback`
  Append y_t to $\mathcal{H}$;
$y^* \leftarrow \mathcal{A}.\texttt{Self\_Verify}(\mathcal{H})$; `// Select final answer via self-verification`
return y^*;

Split	AIME24	GPQA	SWE
Validation	6	32	60
Test	24	166	240

Table E.1: Data size for each split in each dataset.

models are accessed through their respective APIs.[9] Temperature for meta-agent is set to 0.5. For baselines, we strictly use parameters found in original papers and provided code.

Benchmarks Table F.1 shows the detailed statistics for each dataset. We evaluate SWE using its official code available at `https://github.com/SWE-bench/SWE-bench/`.

Baselines For the manual MAS baselines, we adapt the implementation from ADAS. For the automatic MAS baselines, we use the official implementations of ADAS (`https://github.com/ShengranHu/ADAS`), AFlow (`https://github.com/FoundationAgents/MetaGPT/tree/main/examples/aflow`), and MaAS (`https://github.com/bingreeky/MaAS`).

F Standard Deviation for the Experiments

To confirm the statistical significance of the experimental results in Table 1, we repeat the experiment *three* times, following [47, 25].

We can see that the MAS can exhibit high variance due to the inherent nature of multi-agent systems: the variance may be amplified by the interactions among multiple agents [3], and the generated temperature of the agents are typically non-zero.

[9] We use TogetherAI API (`https://www.together.ai/`) for Llama and Qwen.

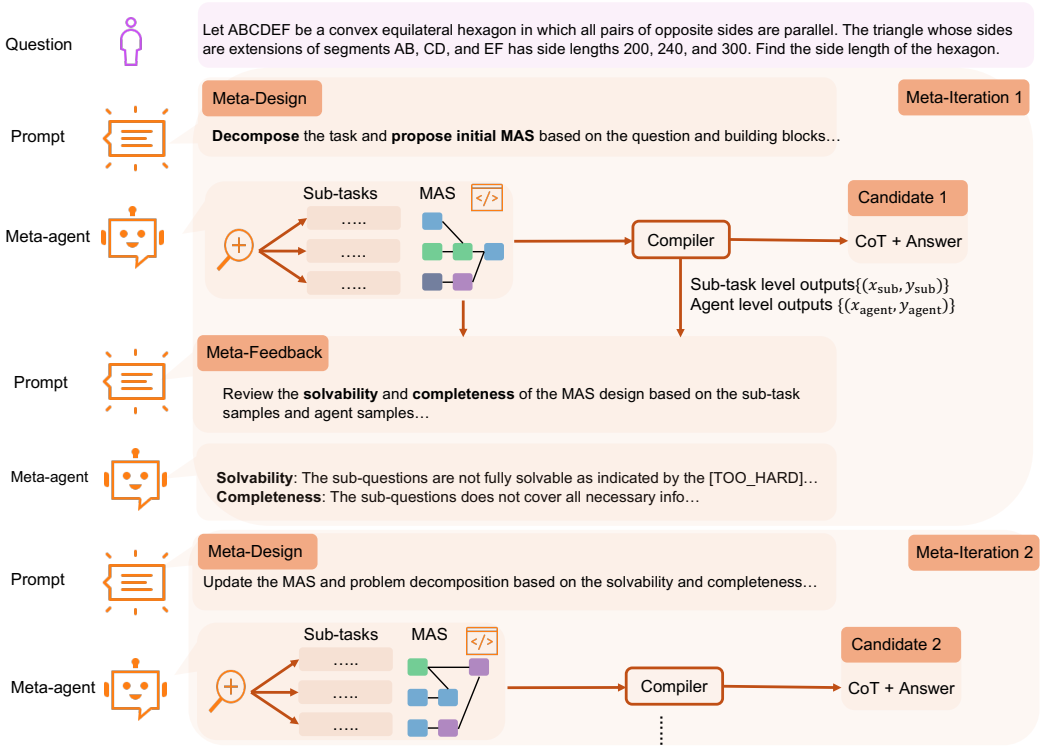

Figure D.1: Example pipeline for the *meta-iterations* in MAS-ZERO.

LLMs	GPT-4o	Llama3.3	Qwen2.5
CoT	±1.97	±1.96	±0.00
CoT-SC	±3.40	±5.20	±1.96
Debate	±7.08	±7.08	±5.20
Self-Refine	±3.93	±1.97	±1.97
ReConcile	±1.97	±1.96	±1.97
MAS-ZERO	±5.89	±3.15	±5.20

Table F.1: Standard deviations of MAS-ZERO and manual MAS in AIME24.

G Prompt Details

In Section 3.1 and Figure D.1, we use prompts to implement meta-design and meta-feedback. Here, we provide more details of the prompts in Figures G.1 and G.2. As mentioned in Appendix C, we merge meta-feedback and meta-design in meta-iterations into a single `Meta_Feedback_Update` function. As a result, there are two sets of prompts: one for the initial `Meta_Design`, and another for `Meta_Feedback_Update` used in the meta-iterations.

H Code Template

In Section 3.1, we use a code template to constrain MAS code generation to filling in a specific `forward` function. Figure H.1 shows how the utility code is provided. Figures H.2, H.3, H.4, and H.5 show the implementations of each building blocks.

Simplified Prompt for `Meta_Design`

Overview You are an expert machine learning researcher testing various agentic systems. Given a set of architectures in the archive and the question. Note that architecture can contain multiple agents, and agent mean a LLM that use for special objectives by specialized setting (instruction, temperature...)
Your objective is to
(1) Perform task decomposition. Specifically, decompose the give question significantly so that the sub-architecture (or node or block) can perform each of the sub-tasks. The output should be sub-task 1, sub-task 2, ... sub-task n. Do not solve the task for the sub-architecture and do not leak the expected answer in your sub-task description/instruction/question (a short-cut like 'output exactly the following...' is also leakage and should be avoided). Instead, decompose the task that easy enough for the sub-architecture to solve. You need to justify how these sub-tasks can achieve the final answer to the orginal questions.
Make sure

1. Include sub-task ID and 'Based on (task i)' in the instruction of a sub-task. This helps each sub-task connects to its prerequisite sub-tasks so that there is enough information to solve it.
2. Each sub-task should be specific and detailed enough to solve and to help achieve the final answer to the given question. The output should be helpful to solve the next sub-task. You need to include details steps (but not the answer) to the sub-task
3. The answer to the last sub-task should be the same as the answer to the final question, so that the architecture successfully solve the complex question by solving each of the sub-task.

(2) Given the resulting sub-task 1, sub-task 2, ... sub-task n, design connections between existing blocks to address each of them. You should structure the architecture as a multi-layered network. Each existing architecture (or blocks) serves as a node, while connections between them act as edges, forming a structured hierarchy of interactions. Additionally, you must determine the number of layers in the network.
For example, if the existing architectures are 'COT, COT_SC, Self-Refine, LLM_debate' and you determine that there can be 3 layers. There are 3 resulting sub-task from (1) sub-task 1, sub-task 2, sub-task 1, sub-task 3:
Example Setup
Resulting sub-tasks: sub-task 1, sub-task 2, sub-task 3, sub-task 4
Available architectures: COT, COT_SC, Self-Refine, LLM_debate
Network with 3 Layers:
Layer 1: COT COT_SC Self-Refine LLM_debate
Layer 2: COT COT_SC Self-Refine LLM_debate
Layer 3: COT COT_SC Self-Refine LLM_debate
Connection Strategies:

1. Linear Connection: Directly link two block to pass information forward.Example: [COT] (address sub-task 1) -> [LLM_debate] (address sub-task 2) (Single connection and exit)
2. Multi-Layer Connection: A block can appear in multiple layers, forming deeper reasoning structures. Example: [COT] (address sub-task 1) -> [LLM_debate] (address sub-task 2) -> [COT -> Self-Refine] (address sub-task 3) (COT appears in both Layer 1 and Layer 3) (the whole [COT -> Self-Refine] is a sub-task architecture that aims to address sub-task 3)

IMPORTANT:

1. Decomposition itself should not be included in the architecture as the question has been decomposed at step (1). Do not assign one block to perform all the sub-tasks (if you put all decomposed sub-tasks into a single instruction for an block, it is very wrong). Instead, assign different block to address each of the sub-task instead.
2. If your previous attempts in the discovered architecture archive are incorrect (fitness value equals to 0), it means the sub-tasks are still too difficult to the corresponding blocks. Please further decompose the question to easier sub-tasks.

Your aim is to design an optimal block connection that can perform well on each of the sub-task. Your code should implement the existing blocks given in the archive (the 'code' entry of blocks) as it-is without medication: Do not propose new blocks or modify existing ones and only change the connections between the given blocks, but block setting like instruction, temperature are allowed to modify.

Figure G.1: Simplified prompt for `Meta_Design`.

Simplified Prompt for `Meta_Feedback_Update`

Carefully review the proposed new architectures ("code" entry), the answer of each sub-tasks ("sub_tasks" entry) , the answer of each agnets ("agents" entry), the final response ("final_response" entry), and the 'memory' (previous final answer extracted from their reponse and the corresponding fitness score, in the format of a list of dictionary final answer: fitness) in all the history user and assistant answers. Reflect on the following points:"

1. **Solvable**: Assess whether all sub-tasks are solvable by the corresponding block via checking the output answer of each sub-task.
   (a) if the answer of the sub-task explicitly contain [TOO_HARD]. This clearly state that task is too hard, which need to be further decomposed. Consider the suggestion given after the [TOO_HARD] (you can see the 'Suggestions:' next to the [TOO_HARD]) and improve your decomposition accordingly.
   (b) If the sub-task answer is incorrect. That means it is not solvable, which need to be improved. It may because
      i. the task is still too difficult for the block, then the sub-task need to be further decomposed.
      ii. some agents in the block is malfunctional or the underlying LLM is too weak to solve the sub-task alone. This can be determined by checking the agents output to decide whether it works as expected. If this is the case, then we need to get rid of the block and use another block in the architecture. There are then two possibilities (1) the agent in the block is not optimal to solve the sub-task, setting needed to be improved (instruction, temperature...) (2) the agent architecture in the block is not optimal, a new block that combine existing blocks in a different way or different settings need to be proposed

   Please **justify** it is (i), the decomposition issue or (ii) the block and agent issue. It could also be both. When proposing new sub-task, **make sure** (1) it is specific and detailed enough to solve and to help achieve the final answer to the given question. (2) all information required to answer the question is provided by the previous answers or the instruction. (3) the related sub-tasks thinking and answers have correctly input to the current sub-task by adding it to the taskInfo list when calling the agent. (4) The output should be helpful to solve the next sub-task. Also make sure the sub-task connection is clearly by clearly state 'Based on the output of sub-task i..' in the sub-task instruction
2. **Completeness** Are the sub-tasks include all neccessay information from the irginal query that can ensure the aggregation of sub-task responses can effectively yild a comprehensive answer to the user query? Note that while a sub-task might include only part of the neccessary information, it is not allowable for any particular piece of critical information to be omitted from all sub-tasks. Make sure the sub-task are connected to the prerequisite sub-tasks so that there is enough information to solve it.

Now, you need to improve or revise the implementation, or implement the new proposed architecture based on the reflection.

Figure G.2: Simplified prompt for `Meta_Feedback_Update`.

Utility Code

```
# Named tuple for holding task information
Info = namedtuple('Info', ['name', 'author', 'content', 'prompt', '
    sub_tasks', 'agents', 'iteration_idx'])

# Format instructions for LLM response
FORMAT_INST = lambda request_keys: f"Reply EXACTLY with the
    following JSON format.\n{str(request_keys)}\nDO NOT MISS ANY
    FIELDS AND MAKE SURE THE JSON FORMAT IS CORRECT!\n"

# Description of the role for the LLM
ROLE_DESC = lambda role: f"You are a {role}."

class LLMAgentBase():

    def __init__(self, output_fields: list, agent_name: str,
                 role='helpful assistant', model=None, temperature=
                     None) -> None:
        self.output_fields = output_fields
        self.agent_name = agent_name

        self.role = role
        self.model = model
        self.temperature = temperature
        # give each instance a unique id
        self.id = random_id()


    def generate_prompt(self, input_infos, instruction) -> str:
        # generate prompt based on the input_infos
        # ...

    def query(self, input_infos: list, instruction, iteration_idx
        =-1) -> dict:
        # call generate_prompt and the LLM to get output
        # ...

    def __repr__(self):
        return f"{self.agent_name} {self.id}"



class AgentArchitecture:
    """
    Fill in your code here.

    def forward(self, taskInfo) -> Union[Info, str]:
        Args:
        - taskInfo (Info): Task information.

        Returns:
        - Answer (Info): Your FINAL Answer.
    """
```

Figure H.1: Utility code.

Implementation of building blocks (CoT)

```
def forward(self, taskInfo):
    # Instruction for the Chain-of-Thought (CoT) approach
    # It is an important practice that allows the LLM to think step
        by step before solving the task.
    cot_instruction = self.cot_instruction

    # Instantiate a new LLM agent specifically for CoT
    # To allow LLM thinking before answering, we need to set an
       additional output field 'thinking'.
    cot_agent = LLMAgentBase(['thinking', 'answer'], 'Chain-of-
       Thought Agent',  model=self.node_model, temperature=0.0)

    # Prepare the inputs for the CoT agent
    # The input should be a list of Info, and the first one is
       often the taskInfo
    cot_agent_inputs = [taskInfo]

    # Get the response from the CoT agent
    thinking, answer = cot_agent(cot_agent_inputs, cot_instruction)
    final_answer = self.make_final_answer(thinking, answer)

    # Return only the final answer
    return final_answer
```

Figure H.2: Implementation of building blocks (CoT)

Implementation of building blocks (CoT-SC)

```
def forward(self, taskInfo):
    # Instruction for step-by-step reasoning
    cot_instruction = self.cot_instruction
    N = self.max_sc # Number of CoT agents

    # Initialize multiple CoT agents with a higher temperature for
        varied reasoning
    cot_agents = [LLMAgentBase(['thinking', 'answer'], 'Chain-of-
        Thought Agent',  model=self.node_model, temperature=0.5) for
         _ in range(N)]

    # Majority voting function to select the most common answer
    from collections import Counter
    def majority_voting(answers):
        return Counter(answers).most_common(1)[0][0]

    thinking_mapping = {}
    answer_mapping = {}
    possible_answers = []
    for i in range(N):
        thinking, answer = cot_agents[i]([taskInfo],
            cot_instruction)
        possible_answers.append(answer.content)
        thinking_mapping[answer.content] = thinking
        answer_mapping[answer.content] = answer

    # Ensembling the answers from multiple CoT agents
    answer = majority_voting(possible_answers)

    thinking = thinking_mapping[answer]
    answer = answer_mapping[answer]

    final_answer = self.make_final_answer(thinking, answer)

    return final_answer
```

Figure H.3: Implementation of building blocks (CoT-SC)

Implementation of building blocks (Debate)

```
def forward(self, taskInfo):
    # Instruction for initial reasoning
    debate_initial_instruction = self.cot_instruction

    # Instruction for debating and updating the solution based on
        other agents' solutions
    debate_instruction = "Given solutions to the problem from other
         agents, consider their opinions as additional advice.
        Please think carefully and provide an updated answer. Put
        your thinking process in the 'thinking' field and the
        updated answer in the 'answer' field. "

    # Initialize debate agents with different roles and a moderate
        temperature for varied reasoning
    debate_agents = [LLMAgentBase(['thinking', 'answer'], 'Debate
        Agent',  model=self.node_model, role=role, temperature=0.5)
        for role in self.debate_role]

    # Instruction for final decision-making based on all debates
        and solutions
    final_decision_instruction = "Given all the above thinking and
        answers, reason over them carefully and provide a final
        answer. Put your thinking process in the 'thinking' field
        and the final answer in the 'answer' field."
    final_decision_agent = LLMAgentBase(['thinking', 'answer'], '
        Final Decision Agent',  model=self.node_model, temperature
        =0.0)

    max_round = self.max_round # Maximum number of debate rounds
    all_thinking = [[] for _ in range(max_round)]
    all_answer = [[] for _ in range(max_round)]

    # Perform debate rounds
    for r in range(max_round):
        for i in range(len(debate_agents)):
            if r == 0:
                thinking, answer = debate_agents[i]([taskInfo],
                    debate_initial_instruction)
            else:
                input_infos = [taskInfo] + [all_thinking[r-1][i]] +
                     all_thinking[r-1][:i] + all_thinking[r-1][i+1:]
                thinking, answer = debate_agents[i](input_infos,
                    debate_instruction)
            all_thinking[r].append(thinking)
            all_answer[r].append(answer)

    # Make the final decision based on all debate results and
        solutions
    thinking, answer = final_decision_agent([taskInfo] +
        all_thinking[max_round-1] + all_answer[max_round-1],
        final_decision_instruction)
    final_answer = self.make_final_answer(thinking, answer)

    return final_answer
```

Figure H.4: Implementation of building blocks (debate)

Implementation of building blocks (Self-refine)

```
def forward(self, taskInfo):
    # Instruction for initial reasoning
    cot_initial_instruction = self.cot_instruction

    # Instruction for reflecting on previous attempts and feedback
        to improve
    cot_reflect_instruction = "Given previous attempts and feedback
        , carefully consider where you could go wrong in your latest
         attempt. Using insights from previous attempts, try to
        solve the task better."
    cot_agent = LLMAgentBase(['thinking', 'answer'], 'Chain-of-
        Thought Agent', model=self.node_model, temperature=0.0)

    # Instruction for providing feedback and correcting the answer
    critic_instruction = "Please review the answer above and
        criticize on where might be wrong. If you are absolutely
        sure it is correct, output exactly 'True' in 'correct'."
    critic_agent = LLMAgentBase(['feedback', 'correct'], 'Critic
        Agent', model=self.node_model, temperature=0.0)

    N_max = self.max_round # Maximum number of attempts

    # Initial attempt
    cot_inputs = [taskInfo]
    thinking, answer = cot_agent(cot_inputs,
        cot_initial_instruction, 0)

    for i in range(N_max):
        # Get feedback and correct status from the critic
        feedback, correct = critic_agent([taskInfo, thinking,
            answer], critic_instruction, i)
        if correct.content == 'True':
            break

        # Add feedback to the inputs for the next iteration
        cot_inputs.extend([thinking, answer, feedback])

        # Reflect on previous attempts and refine the answer
        thinking, answer = cot_agent(cot_inputs,
            cot_reflect_instruction, i + 1)

    final_answer = self.make_final_answer(thinking, answer)

    return final_answer
```

Figure H.5: Implementation of building blocks (self-refine)