Unified-MAS: Universally Generating Domain-Specific Nodes for Empowering Automatic Multi-Agent Systems

Hehai Lin[♠], **Yu Yan**[♠], **Zixuan Wang**[♠], **Bo Xu**[♠], **Sudong Wang**[♠], **Weiquan Huang**[♠], **Ruochen Zhao**[◇], **Minzhi Li**[♡♣], **Chengwei Qin**[♠] *
[♠]The Hong Kong University of Science and Technology (Guangzhou)
[◇]Singapore University of Technology and Design [♡]National University of Singapore
[♣]Institute for Infocomm Research (I^2R), A*STAR

Project page Gallery & Live Demo Code

Abstract

Automatic Multi-Agent Systems (MAS) generation has emerged as a promising paradigm for solving complex reasoning tasks. However, existing frameworks are fundamentally bottlenecked when applied to knowledge-intensive domains (e.g., healthcare and law). They either rely on a static library of general nodes like Chain-of-Thought, which lack specialized expertise, or attempt to generate nodes on the fly. In the latter case, the orchestrator is not only bound by its internal knowledge limits but must also simultaneously generate domain-specific logic and optimize high-level topology, leading to a severe architectural coupling that degrades overall system efficacy. To bridge this gap, we propose Unified-MAS that decouples granular node implementation from topological orchestration via offline node synthesis. Unified-MAS operates in two stages: (1) **Search-Based Node Generation** retrieves external open-world knowledge to synthesize specialized node blueprints, overcoming the internal knowledge limits of LLMs; and (2) **Reward-Based Node Optimization** utilizes a perplexity-guided reward to iteratively enhance the internal logic of bottleneck nodes. Extensive experiments across four specialized domains demonstrate that integrating Unified-MAS into four Automatic-MAS baselines yields a much better performance-cost trade-off, achieving up to a 14.2% gain while significantly reducing costs. Further analysis reveals its robustness across different designer LLMs and its generalizability to general domains such as mathematical reasoning.

1 Introduction

The rapid evolution of Large Language Models (LLMs) has transformed the landscape of Artificial Intelligence (Ferrag et al., 2025; Yu et al., 2026b; Huang et al., 2026). Building upon this foundation, LLM-based Multi-Agent Systems (MAS) have emerged as a powerful paradigm, demonstrating superior capabilities by leveraging collaborative intelligence (Lin et al., 2025; Wu et al., 2025). Traditionally, designing effective MAS required meticulous manual engineering by human experts (Wang et al., 2022; Shinn et al., 2023). Recently, the community has experienced a paradigm shift towards automatic MAS generation (Ye et al., 2025; Tran et al., 2025). By utilizing techniques such as graph neural networks or code-based optimization, Automatic-MAS can discover novel agentic workflows that often surpass human-designed solutions on general-purpose benchmarks (Ke et al., 2025a).

Despite these advancements, a significant limitation persists: the severe performance degradation of Automatic-MAS in *specialized, knowledge-intensive domains* (Hong et al., 2023; Xu et al., 2025). As illustrated in Figure 1(a), our preliminary study reveals that when applied to domains requiring specialized expertise (e.g., legal judgment or clinical diagnosis), they consistently underperform compared to manually crafted, domain-specific MAS. This performance gap stems from the fact that most Automatic-MAS rely on a static set of general-purpose nodes like Chain-of-Thought (CoT) and Debate (Du et al., 2024). Lacking specialized priors, the orchestrator tends to merely stack general nodes, failing to capture the nuanced requirements for expert-level tasks (Li et al., 2024).

Recent works have attempted to explore dynamic node generation, prompting the orchestrator to invent new sub-agents on the fly (Zhang et al., 2025b; Ruan et al., 2026). However, these approaches suffer from two fundamental flaws. First, they are bound by the *internal knowledge limits* of the LLM. Without grounding in external, domain-specific data (e.g., legal statutes or clinical protocols), the

*Corresponding to chengweiqin@hkust-gz.edu.cn

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 44511–44541
October 24-29, 2026 ©2026 Association for Computational Linguistics

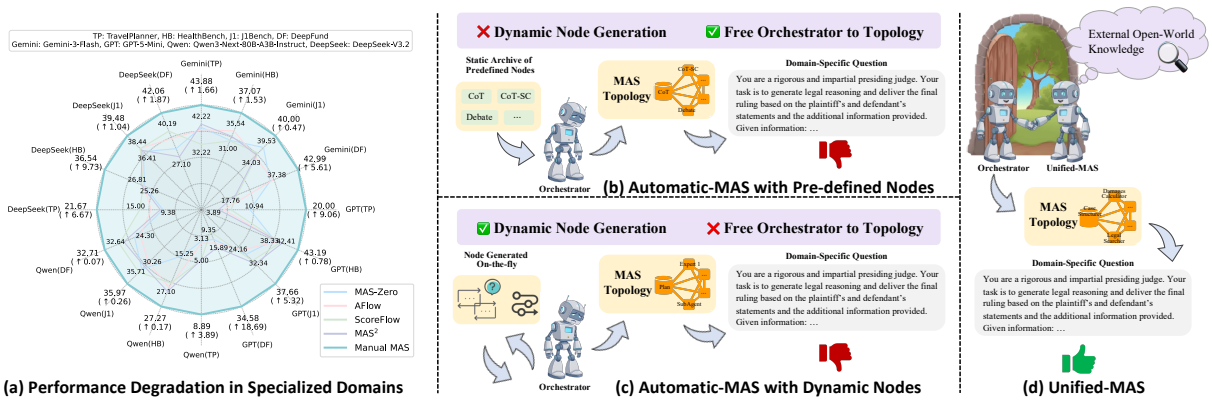

Figure 1: Overview of MAS paradigms. (a) Performance degradation in specialized domains, where Automatic-MAS with predefined nodes underperforms manual MAS. (b)-(c) Comparison of existing Automatic-MAS paradigms, illustrating the dichotomy between dynamic node generation and topological flexibility. (d) Unified-MAS leverages open-world knowledge to generate domain-specific nodes, effectively empowering existing Automatic-MAS.

LLM inevitably hallucinates superficial or erroneous node logic (Huang et al., 2025; Yu et al., 2026a). Second, it introduces a severe *architectural coupling*. Burdening the orchestrator with the granular implementation of micro-level domain logic distracts and dilutes its primary capability: managing macro-level topology (Ke et al., 2026).

To address these challenges, we propose **Unified-MAS**, a novel framework that advocates for the decoupling of granular node implementation from topological orchestration. As an offline synthesizer, Unified-MAS generates domain-specific nodes for any domain that can be seamlessly integrated into any existing Automatic-MAS. Specifically, Unified-MAS contains two stages: (1) **Search-Based Node Generation**: Unified-MAS first extracts multi-dimensional keywords from task samples and synthesizes targeted queries. Then, to overcome parametric knowledge limitations, it retrieves external open-world knowledge across diverse sources (i.e., Google, GitHub, and Google Scholar) to distill domain-specific design principles, generating an initial set of specialized nodes. (2) **Reward-Based Node Optimization**: Initially generated nodes, while functionally relevant, are often coarse-grained and logically brittle, which may trigger compounding errors in a multi-agent scenario. We introduce a node optimization mechanism driven by a perplexity-guided reward. By quantifying the stability and magnitude of reasoning progress contributed by each node, Unified-MAS identifies *bottleneck nodes* and iteratively refines their internal implementation (e.g., refining prompt constraints or adding necessary sub-agent calls).

We comprehensively evaluate Unified-MAS on four highly specialized benchmarks, i.e., TravelPlanner for constrained travel planning (Xie et al., 2024), HealthBench for healthcare (Arora et al., 2025), J1Bench for legal judgment (Jia et al., 2025), and DeepFund for financial decision-making (Li et al., 2025). We integrate the generated nodes into four general Automatic-MAS baselines, MAS-Zero (Ke et al., 2025b), AFlow (Zhang et al., 2024), ScoreFlow (Wang et al., 2025b), and MAS^2 (Wang et al., 2025a), and evaluate the system with four different LLMs as orchestrators. The evaluations reveal several key findings: (1) *Dual Advantage in Performance and Cost.* Unified-MAS consistently drives performance gains, achieving up to a 14.2% gain, while simultaneously reducing costs. This underscores the critical role of domain-specific priors, positioning our framework as a universal catalyst for elevating general Automatic-MAS into expert-level systems. (2) *Strong Robustness and Generalizability.* Unified-MAS not only exhibits robust performance across various designer LLMs but also generalizes seamlessly to general domains like mathematics. (3) *Efficacy of Perplexity-Guided Optimization.* The synthesized nodes progressively improve through reward-based optimization, effectively strengthening their logical reliability in complex domains. Our main contributions are summarized as follows:

- We identify the limitations of Automatic-MAS in specialized domains and propose a new paradigm that *decouples* granular node implementation from topology orchestration.
- We propose Unified-MAS, which leverages external retrieval to synthesize specialized

nodes, and employs perplexity-guided reward optimization to improve their internal logic.

- Our extensive experiments demonstrate that Unified-MAS consistently improves the performance of existing Automatic-MAS while reducing costs across complex domains.

2 Related Work

2.1 Automatic-MAS with Pre-defined Nodes

The most prevalent methods construct Multi-agent Systems (MAS) using a static archive of pre-defined nodes, which consists of manually designed structures, such as CoT, CoT-SC (Wang et al., 2022), and self-reflection (Madaan et al., 2023; He et al., 2025), where each node functions as an agent. The orchestrator's role is to determine the optimal topological architecture between these nodes (Chen et al., 2024; Lin et al., 2026).

Inference-time approaches rely on sophisticated prompting and iterative search without updating model weights (Jwalapuram et al., 2026). For example, ADAS represents the MAS as code and iteratively generates new architectures using a Meta Agent Search on a validation set (Hu et al., 2024). AFlow employs Monte Carlo Tree Search (MCTS) to discover effective agentic workflows (Zhang et al., 2024), while DyLAN enables multi-round interactions with dynamic agent selection and early-stopping mechanisms to enhance efficiency (Liu et al., 2023). MAS-Zero introduces a self-reflective feedback loop, allowing the orchestrator to optimize the MAS without requiring an external validation set (Ke et al., 2025b). **Training-time** approaches optimize the orchestrator to generate high-quality MAS in one-shot by learning from generated trajectories. ScoreFlow utilizes Score-DPO, a variant of direct preference optimization, to incorporate quantitative feedback into the orchestrator's training (Wang et al., 2025b). MAS^2 learns a self-generative, self-configuring, and self-rectifying workflow (Wang et al., 2025a), while MAS-Orchestra models MAS construction as a function-calling task optimized via Group Relative Policy Optimization (GRPO) (Ke et al., 2026). However, a critical limitation of these methods is their reliance on a static set of general-purpose nodes. As demonstrated in Figure 1, when applied to specialized domains, their performance often lags behind manually crafted domain-specific MAS due to the lack of expert knowledge.

2.2 Automatic-MAS with Dynamic Nodes

To address the rigidity of pre-defined archives, recent community has turned to dynamic node generation, where the orchestrator attempts to introduce new nodes on the fly based on task requirements. MetaAgent first identifies and implements necessary nodes before optimizing the system using Finite State Machines (Zhang et al., 2025b). EvoAgent serves as a generic method to automatically extend expert agents into MAS via evolutionary algorithms (Yuan et al., 2025). Similarly, Aorchestra abstracts nodes into a tuple of $\langle$*Instruction, Context, Tools, Model*$\rangle$, enabling the orchestrator to dynamically populate these slots following task decomposition (Ruan et al., 2026). While promising, these approaches are constrained by the orchestrator's internal knowledge. If the necessary domain expertise is absent during the orchestrator's pre-training, the system is prone to hallucinations, resulting in ineffective or erroneous nodes (Ji et al., 2023). Furthermore, recent observations suggest that an effective orchestrator should prioritize architectural connectivity rather than the granular implementation of individual nodes (Ke et al., 2026).

In this paper, we introduce Unified-MAS, a two-stage workflow designed to generate domain-specific nodes, which can be seamlessly integrated into existing Automatic-MAS frameworks. This integration injects essential domain knowledge into the system while liberating the orchestrator from the burden of node design, thereby allowing it to fully leverage its search capabilities to optimize the topological structure of the MAS.

3 Methodology

As illustrated in Figure 2, Unified-MAS introduces a new paradigm by acting as an offline node synthesizer prior to the Automatic-MAS topological search. This design bridges the gap between general automatic orchestration and domain specificity through a highly decoupled two-stage pipeline: (1) Search-Based Node Generation, which overcomes parametric knowledge limits, and (2) Reward-Based Node Optimization, which improves the internal reasoning logic of individual nodes.

3.1 Problem Formulation

Existing Automatic-MAS approaches typically frame the system design as a search problem over a topology space Ω using a static library of prede-

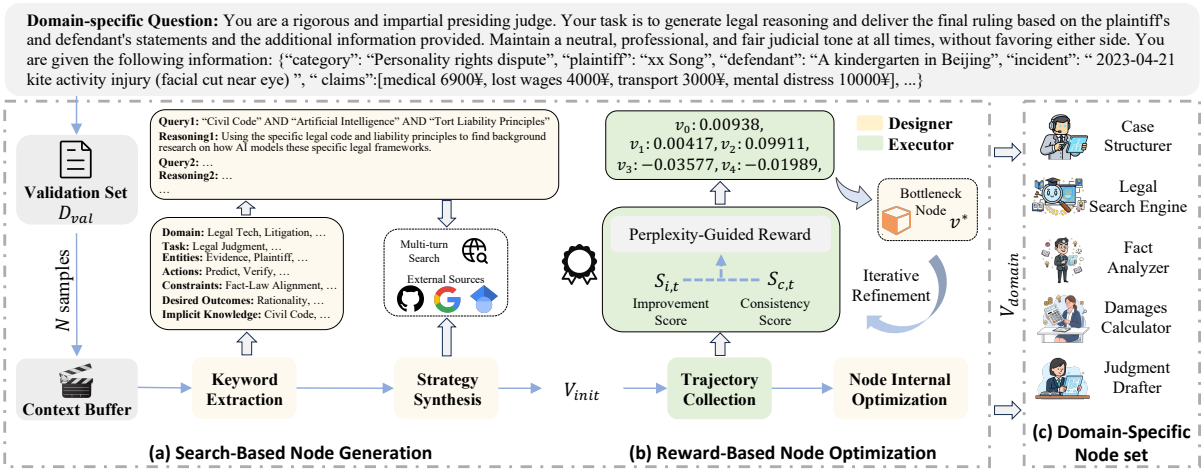

Figure 2: Illustration of Unified-MAS. (a) Search-Based Node Generation retrieves external knowledge via keyword-strategy driven queries to initialize V_{init}. These nodes are subsequently fed into (b) Reward-Based Node Optimization, which iteratively identifies and refines bottleneck nodes guided by a perplexity-based reward. Finally, Unified-MAS generates (c) a domain-specific node set, which can be integrated into existing Automatic-MAS.

fined, general-purpose nodes $\mathcal{V}_{fix}$. Let $\mathcal{M}$ represent a MAS configuration defined by its topological structure $\mathcal{G} \in \Omega$ and the selection of functional nodes $V \subseteq \mathcal{V}_{fix}$. The objective is to identify the optimal configuration $\mathcal{M}^*$ that maximizes the performance $\mathcal{R}$ over the data distribution $\mathcal{D}$:

$$\mathcal{M}^* = \underset{\mathcal{G}\in\Omega, V\subseteq\mathcal{V}_{fix}}{\arg\max} \ \mathbb{E}_{x\sim\mathcal{D}}[\mathcal{R}(\mathcal{M}(x;\mathcal{G},V))] \quad (1)$$

This formulation inherently limits the solution space to combinations of generic reasoning nodes in $\mathcal{V}_{fix}$. Unified-MAS addresses this limitation by expanding the search space from a static $\mathcal{V}_{fix}$ to a dynamically domain-adaptive set $\mathcal{V}_{domain}$.

3.2 Search-Based Node Generation

Multi-Dimensional Keyword Extraction. To construct $\mathcal{V}_{domain}$, we first sample N examples from a validation set $\mathcal{D}_{val}$ to form a context buffer $\mathcal{C}$. We prompt the LLM to analyze $\mathcal{C}$ and extract keywords across seven dimensions. This granular decomposition ensures no critical aspect of the domain is overlooked. (1) *Domain*: the macro-industry context (e.g., Fintech); (2) *Task*: the core technical problem (e.g., decision-making); (3) *Entities*: the specific data entities such as company news; (4) *Actions*: the operations or methods performed on these entities; (5) *Constraints*: task requirement such as low latency; (6) *Desired Outcomes*: the target metrics (e.g., accuracy); and (7) *Implicit Knowledge*: latent expert intuitions that are not explicitly stated but are essential for success.

Strategy-Driven Query Synthesis. We then synthesize these seven dimensions into four targeted search strategies, each designed to retrieve a specific layer of system design knowledge: (1) Strategy A (**Background Knowledge**): combining *Domain* and *Implicit Knowledge* to retrieve background information and survey papers; (2) Strategy B (**System Architecture**): combining *Task* and *Constraints* to search for architectural patterns that satisfy specific requirements; (3) Strategy C (**Code Implementation**): combining *Entities* and *Actions* to locate repositories for libraries handling specific data types from GitHub; and (4) Strategy D (**Evaluation**): combining *Task* and *Desired Outcomes* to identify standard benchmarks and evaluation metrics for this specific domain.

Knowledge Aggregation and Node Generation. Finally, we perform multi-turn search (Zhao et al., 2026) using appropriate search engines, and aggregate the retrieved content into strategy-specific summaries. Based on these summaries and guided by a node generation prompt, the LLM generates an initial node set $\mathcal{V}_{init} = \{v_1, \ldots, v_m\}$, where each node v_i represents a domain-specific agent including its system prompts and tool specifications.

3.3 Reward-Based Node Optimization

Although the initial nodes in $\mathcal{V}_{init}$ successfully capture essential domain priors, possessing knowledge does not equal robust reasoning. The preliminary nature of their generation often leaves their internal implementation superficial, struggling to handle the

nuanced logic required for expert-level tasks. Without iterative refinement, these unstable reasoning mechanics can easily bottleneck the overall system efficacy. Therefore, to transition these nodes from coarse blueprints into reliable operators, we formulate MAS execution as a trajectory reasoning, assign a reward for each node, and optimize the *bottleneck node* with the lowest reward.

Although some nodes are logically parallel, their outputs can be treated as being sequentially appended to the MAS output during execution. Let a reasoning trajectory be a sequence of states $\tau = \{h_0, h_1, \ldots, h_m\}$ generated by the sequential execution of nodes $\{v_1, \ldots, v_m\}$. Here, h_0 represents the empty context before any node execution, while h_t (for $t \geq 1$) denotes the output generated by node v_t. The accumulated context after executing node v_t is defined as the concatenation of all preceding outputs: $A_t = [h_0, h_1, \ldots, h_t]$.

Dynamically generated subtasks rarely have gold-standard intermediate labels, making it difficult to obtain reliable dense rewards for individual sub-agents. Terminal correctness offers only sparse, execution-level supervision and cannot indicate whether a given node advances or detracts from the reference solution. We therefore use perplexity as a proxy reward. Specifically, to evaluate the effectiveness of each node, we measure how well the accumulated reasoning trajectory predicts the ground-truth answer y. We compute the perplexity of generating y given the input question q and the accumulated context A_t under an LLM P_θ:

$$\text{PPL}(y|q, A_t) = \exp(-\frac{1}{|y|}\sum_{j=1}^{|y|} \log P_\theta(y_j|q, A_t)) \tag{2}$$

Based on this definition, we derive an objective function $\mathcal{J}$ by maximizing the negative log-perplexity, which reflects the predictability of the answer y given the accumulated reasoning steps:

$$\begin{aligned}\mathcal{J}(P_\theta, y, q, A_t) &= -\log(\text{PPL}(y|q, A_t)) \\ &= \frac{1}{|y|}\sum_{j=1}^{|y|} \log P_\theta(y_j|q, A_t)\end{aligned} \tag{3}$$

A higher $\mathcal{J}$ corresponds to lower perplexity, indicating that the sequence of reasoning steps up to node v_t has effectively reduced the model's uncertainty and guided the system closer to the correct solution. To standardize evaluation across different queries, we define $\mathcal{J}_0$ as the predictability of the answer using the model's direct inference capability, i.e., with an empty context A_0, $\mathcal{J}_0 = \mathcal{J}(P_\theta, y, q)$.

To optimize nodes based on the objective defined above, we evaluate each node from two complementary perspectives: *utility* and *stability*. An effective node should provide a reasoning path that is not only impactful (yielding a considerable gain) but also consistent (avoiding erratic fluctuations) (Liu et al., 2025b). We therefore introduce two scores to assess the quality of node v_t:

Improvement Score ($\mathcal{S}_{i,t}$) It measures the relative gain in the objective compared to the baseline $\mathcal{J}_0$, reflecting the strength of the node's contribution. Formally,

$$\mathcal{S}_{i,t} = \tanh(\delta(P_\theta, y, q, A_t) + 1) \tag{4}$$

$$\delta(P_\theta, y, q, A_t) = \frac{\mathcal{J}(P_\theta, y, q, A_t) - \mathcal{J}_0}{|\mathcal{J}_0|} \tag{5}$$

where $\delta(P_\theta, y, q, A_t)$ represents the normalized improvement over direct inference. The $\tanh$ function is used to smooth outliers and bound the score.

Consistency Score ($\mathcal{S}_{c,t}$) It assesses the stability of the reasoning process. To measure whether the benefit improves consistently as reasoning depth increases, we compute Kendall's Tau correlation coefficient (Kendall, 1938) between the sequence of objective values $\{\mathcal{J}_1, \ldots, \mathcal{J}_t\}$ and their corresponding step indices. The consistency score is:

$$\mathcal{S}_{c,t} = \frac{2}{t(t-1)} \sum_{1 \leq i,j \leq t}^{i<j} \text{sgn}(\mathcal{J}_i - \mathcal{J}_j) \cdot \text{sgn}(i - j) \tag{6}$$

where $\text{sgn}(\cdot)$ denotes the Signum function. A higher $\mathcal{S}_c$ indicates a more stable reasoning.

The **Node Quality Score ($\mathcal{S}_t$)** is computed as a weighted combination of these two scores:

$$\mathcal{S}_t = (1 - \alpha)\mathcal{S}_{i,t} + \alpha\mathcal{S}_{c,t} \tag{7}$$

where α is a balancing hyperparameter. Based on this score, we define the perplexity-guided reward of node v_t as *the incremental gain in node quality*:

$$r_t = \begin{cases} \mathcal{S}_t - \mathcal{S}_{t-1} & \text{if } t > 1, \\ \mathcal{S}_t & \text{if } t = 1 \end{cases} \tag{8}$$

To refine node implementations, we perform optimization for K epochs on the validation set $\mathcal{D}_{val}$.

In each epoch, we calculate the average reward $\bar{r}(v)$ for each node $v \in \mathcal{V}_{init}$ across all samples of $\mathcal{D}_{val}$. The node with the lowest average reward is identified as the *bottleneck node*:

$$v^* = \arg\min_{v \in \mathcal{V}_{init}} \bar{r}(v) \tag{9}$$

We then retrieve the samples where v^* produces the lowest rewards and use them to refine its internal instructions or add additional LLM calls to maximize future rewards. Importantly, in each epoch, samples for which v^* is not the lowest-reward node are excluded from the optimization process, ensuring targeted and stable refinement.

There are two types of LLMs in Unified-MAS. To distinguish them from the LLMs used in Automatic-MAS (orchestrator), we denote them as Designer and Executor. The Designer is responsible for generating and optimizing domain-specific nodes. We employ Gemini-3-Pro as the default Designer due to its strong capabilities. The effect of different Designer models is further investigated in Section 5.2.1. The Executor executes nodes and collects trajectories to compute the perplexity-guided reward. Considering that this computation requires direct access to token-level logits and the practical deployment, we employ Qwen3-Next-80B-A3B-Instruct as the default Executor.

4 Experimental Settings

Benchmarks and Evaluation Metrics. We select four benchmarks spanning different specialized domains. (1) **TravelPlanner** (Xie et al., 2024) for constrained planning. Performance is measured by accuracy. (2) **HealthBench** (Arora et al., 2025) for health diagnosis. Responses are scored against a rubric using an LLM-Judge. (3) **J1Bench** (Jia et al., 2025) simulates automatic legal adjudication. The agent synthesizes conflicting testimonies to produce a final verdict, evaluated by an LLM-Judge under a unified standard. (4) **DeepFund** (Li et al., 2025) for stock market decision-making and evaluated by accuracy. All metrics are normalized to $[0, 100\%]$. We report the average performance and the average cost (in USD \$). The dataset statistics are provided in Appendix C. We elaborate the retrieval leakage control in Appendix D.4. The LLM-as-a-Judge prompts are cataloged in Appendix F.

Baselines. We adopt three categories of MAS to ensure a comprehensive evaluation. (1) **Specific Manual MAS**: PMC (Zhang et al., 2025a) for TravelPlanner, Diagnosis-MAS (Chen et al., 2025) for HealthBench, Court-MAS (Jia et al., 2025) for J1Bench, and DeepFund-MAS (Li et al., 2025) for DeepFund. These serve as the manual-design performance standard. (2) **Automatic-MAS with Dynamic Nodes**: MetaAgent (Zhang et al., 2025b), EvoAgent (Yuan et al., 2025), and AOrchestra (Ruan et al., 2026), which generate nodes on the fly during problem solving. (3) **Automatic-MAS with Pre-defined Nodes**: We benchmark against leading Automatic-MAS that rely on static nodes, i.e., AFlow (Zhang et al., 2024), MAS-Zero (Ke et al., 2025b), ScoreFlow (Wang et al., 2025b), and MAS^2 (Wang et al., 2025a). We empower these baselines by replacing their general nodes with the Unified-MAS's domain-specific node libraries.

Test Models. We deploy the *same* LLM for every component within the final Automatic-MAS setups for fair comparison. Our evaluation spans four different models, including two closed-source models, Gemini-3-Flash (Team et al., 2023) and GPT-5-Mini (Singh et al., 2025), and two open-source models, Qwen3-Next-80B-A3B-Instruct (Team, 2025) and DeepSeek-V3.2 (Liu et al., 2025a). Key configurations are documented in Appendix D, and prompts for Unified-MAS are listed in Appendix F.

5 Results and Analysis

5.1 Main Results

The Domain Barrier: Manual vs. Automatic-MAS. Table 1 shows that task-specific Manual MAS consistently outperforms Automatic-MAS baselines across nearly all settings. For example, with Gemini-3-Flash, Manual MAS achieves an average score of 40.99, exceeding all Automatic-MAS baselines. This gap highlights the importance of domain expertise in complex tasks. Even with dynamic node generation, general-purpose orchestrators struggle to discover effective topologies without incorporating specialized knowledge.

Trap of Dynamic Node Generation. Methods attempting dynamic node generation (i.e., MetaAgent, EvoAgent, AOrchestra) exhibit flashes of potential but suffer from severe systemic instability. For example, while EvoAgent marginally surpasses Manual MAS on J1Bench (e.g., 41.82 vs. 40.00 with Gemini-3-Flash), these dynamic methods fail catastrophically on TravelPlanner, often performing worse than the Vanilla baseline.

Unified-MAS Improves Performance and Efficiency. As shown in Table 1, integrating the

Method	Gemini-3-Flash						GPT-5-Mini					
	TP	HB	J1	DF	Avg.Perf ↑	Avg.Cost ↓	TP	HB	J1	DF	Avg.Perf ↑	Avg.Cost ↓
Vanilla	38.33	26.36	33.25	16.82	28.69	2.484	3.89	37.84	23.77	12.15	19.41	0.469
Manual MAS	43.88	37.07	40.00	42.99	40.99	21.898	20.00	43.19	37.66	34.58	33.86	17.164
MetaAgent	41.11	29.67	37.40	19.63	31.95	4.116	2.22	39.04	30.39	13.08	21.18	1.443
EvoAgent	41.11	34.13	41.82	37.38	38.61	44.791	3.89	41.40	36.49	12.15	23.48	17.498
AOrchestra	40.00	28.98	34.16	22.43	31.39	6.856	6.67	38.41	30.00	17.76	23.21	3.131
MAS-Zero	40.61	31.30	39.53	35.94	36.85	132.179	10.94	38.33	26.72	12.50	22.12	111.910
+ *Unified*	46.88	33.60	**48.91**	48.44	44.46 +7.61	123.803 -8.376	**23.44**	40.21	30.94	28.12	30.68 +8.56	44.011 -67.899
AFlow	39.44	35.54	34.03	37.38	36.60	32.462	5.00	40.19	24.16	14.02	20.84	7.561
+ *Unified*	48.33	**37.69**	44.29	**54.21**	**46.13** +9.53	32.861 +0.399	14.44	**49.97**	41.82	38.97	**36.30** +15.46	7.734 +0.173
ScoreFlow	32.22	31.00	36.10	18.69	29.50	36.908	6.67	41.57	30.52	9.35	22.03	5.914
+ *Unified*	39.44	32.37	44.55	50.47	41.71 +12.21	29.071 -7.837	7.78	43.37	34.03	**40.19**	31.34 +9.31	5.969 +0.055
MAS^2	42.22	33.07	34.94	17.76	32.00	24.174	3.89	42.41	32.34	15.89	23.63	3.368
+ *Unified*	**48.89**	35.09	46.25	49.06	44.82 +12.82	14.819 -9.355	4.44	46.64	**42.73**	36.89	32.68 +9.05	2.858 -0.510
Method	**Qwen3-Next-80B-A3B-Instruct**						**DeepSeek-V3.2**					
Vanilla	2.22	20.07	27.66	23.36	18.33	0.176	8.33	23.51	31.69	26.17	22.43	0.244
Manual MAS	8.89	27.27	35.97	32.71	26.21	5.867	21.67	36.54	39.48	42.06	34.94	8.149
MetaAgent	1.67	21.22	29.74	10.28	15.73	2.148	0.56	23.84	32.47	28.97	21.46	2.586
EvoAgent	1.67	14.03	38.04	23.36	19.28	6.711	5.00	30.76	40.37	33.64	27.44	8.358
AOrchestra	1.11	20.95	34.81	35.51	23.10	1.613	4.44	26.94	36.88	25.23	23.37	2.198
MAS-Zero	3.13	15.25	34.06	26.56	19.75	29.911	9.38	25.26	36.41	31.25	25.58	36.804
+ *Unified*	9.40	20.90	40.16	33.90	26.09 +6.34	18.939 -10.972	**23.44**	33.64	47.58	39.06	35.93 +10.35	18.622 -18.182
AFlow	3.33	25.81	30.26	24.30	20.93	1.678	12.22	25.56	38.44	36.45	28.17	7.773
+ *Unified*	5.56	32.46	37.40	**56.08**	32.88 +11.95	1.665 -0.013	22.22	**39.51**	50.26	**46.73**	**39.68** +11.51	2.593 -5.180
ScoreFlow	5.00	24.31	35.71	32.64	24.42	4.585	15.00	25.85	38.31	40.19	29.84	7.361
+ *Unified*	10.56	31.55	39.87	53.27	**33.81** +9.39	3.849 -0.736	18.89	34.60	**54.81**	45.79	38.52 +8.68	5.924 -1.437
MAS^2	5.00	27.10	32.47	30.84	23.85	2.000	10.00	26.81	37.27	27.10	25.30	1.650
+ *Unified*	**11.11**	**32.55**	**43.81**	42.99	32.62 +8.77	1.008 -0.992	17.22	32.41	52.86	38.32	35.20 +9.90	1.338 -0.312

Table 1: Quantification comparison of Unified-MAS and baselines on four benchmarks. Rows highlighted in blue indicate methods with domain-specific nodes generated by Unified-MAS. **TP**: TravelPlanner, **HB**: HealthBench, **J1**: J1Bench, **DF**: DeepFund. **Avg.** reports average performance and cost. **Bold** denotes the best result.

domain-specific node set generated by Unified-MAS substantially improves the performance of predefined Automatic-MAS while universally reducing costs. In terms of average performance, incorporating domain-specific nodes yields consistent improvements across all settings, with gains ranging from 6.0% (MAS-Zero with Qwen3-Next-80B-A3B-Instruct) to 14.2% (AFlow with GPT-5-Mini). Figure 3 further demonstrates that methods enhanced by Unified-MAS consistently achieve a superior performance–cost trade-off compared to both manual and unenhanced automatic baselines[1]. By replacing inefficient general nodes with optimized domain-specific nodes, Unified-MAS enables the system to solve complex problems with fewer and more effective steps. These results confirm that Unified-MAS successfully bridges the gap, combining the reliability of expert nodes with the scalability of automated design.

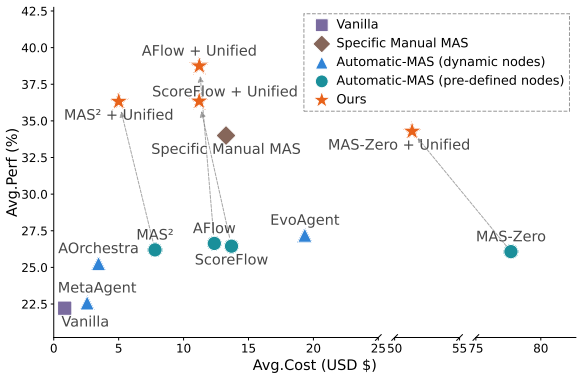

Figure 3: Performance-cost trade-off averaged across four LLMs. Gray arrows illustrate Unified-MAS elevating baselines to higher performance at reduced costs.

Ablation Study. As shown in Figure 4, the results at epoch 0 and 10 correspond to the ablation variant **"Stage 1 only"** and **"Stage 1 + stage 2"**. First, the latter markedly outperforms the former. Second, even "Stage 1 only" already outperforms existing automatic baselines. Therefore, both stages contribute meaningfully to the overall performance. On one hand, in real-world scenarios where labeled validation data is unavailable, Unified-MAS can still be directly and effectively

[1]The difference between online inference cost and offline generation cost is detailed in Appendix D.3.

Method	Designer	GPT-5-Mini		DeepSeek-V3.2	
		Perf ↑	Cost ↓	Perf ↑	Cost ↓
AFlow	-	20.84	7.561	28.17	7.773
	Gemini-3-Pro	**36.30**	7.734	**39.68**	2.593
+ *Unified*	Gemini-3-Flash	33.46	**4.421**	37.14	2.221
	GPT-5-Mini	35.84	7.093	35.49	**2.125**
MAS^2	-	23.63	3.368	25.30	1.650
	Gemini-3-Pro	**32.68**	**2.858**	**35.20**	**1.338**
+ *Unified*	Gemini-3-Flash	30.04	4.713	32.03	1.615
	GPT-5-Mini	30.57	5.987	32.55	2.204

Table 2: Robustness across different Designer LLMs.

Method	GPT-5-Mini		DeepSeek-V3.2	
	Perf ↑	Cost ↓	Perf ↑	Cost ↓
Vanilla	55.10	0.234	42.86	0.063
MAS-Zero	57.14	30.546	48.98	12.162
+ *Unified*	59.18 +2.04	19.351 -11.195	53.06 +4.08	8.899 -3.263
AFlow	59.18	1.736	48.98	0.472
+ *Unified*	67.35 +8.17	3.209 +1.473	55.10 +6.12	0.718 +0.246
ScoreFlow	57.14	0.701	44.90	0.305
+ *Unified*	61.22 +4.08	0.854 +0.153	57.14 +12.24	0.462 +0.157
MAS^2	63.27	1.133	51.02	0.434
+ *Unified*	67.35 +4.08	1.040 -0.093	66.67 +15.65	0.884 +0.450

Table 3: Results of General Automatic-MAS with/without Unified-MAS on AIME24&25.

deployed using only the first stage. On the other hand, Stage-1 nodes can also be further improved by label-free self-evolution techniques, such as self-consistency, reflection, LLM-judge feedback, or online execution feedback, without being strictly limited to ground-truth validation labels. More ablation studies can be found in Appendix D.7.

5.2 Further Analysis

5.2.1 Robustness to Designer Choices

Table 2 reveals that Unified-MAS universally elevates baseline performance across all three Designers, demonstrating that Unified-MAS is highly robust to the choice of the "Designer LLM". Interestingly, we observe an architectural divergence based on the LLM's inherent preferences (Appendix E). Gemini models tend to synthesize concise, macro-level workflows (5-6 nodes), whereas GPT-5-Mini prefers micro-level granularity (about 10 nodes by decomposing complex nodes further). Despite these distinct topological preferences, Unified-MAS is not bottlenecked by any single LLM, consistently driving substantial performance gains.

5.2.2 Generalizability to General Domains

While our main evaluation focuses on specialized domains, Table 3 extends the analysis to general domains (mathematical reasoning) using AIME

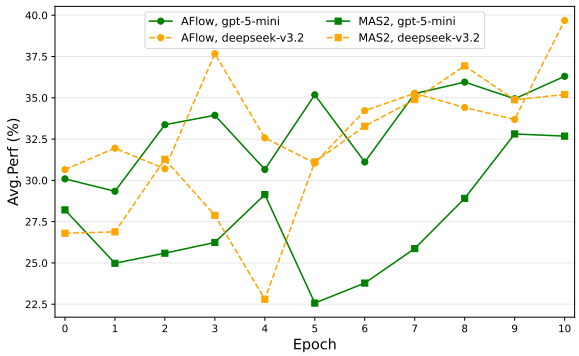

Figure 4: Epoch-wise performance dynamics during node optimization using Gemini-3-Pro as the Designer.

2024 and 2025 (MAA-Committees, 2025). Integrating Unified-MAS consistently improves performance across all baselines for both GPT-5-Mini and DeepSeek-V3.2. Although the gains are more modest than the substantial improvements observed in knowledge-intensive tasks, the results prove that our framework can successfully synthesize reasonable, fine-grained mathematical nodes (see Appendix E), demonstrating broad applicability even in conventional reasoning tasks. We further discuss the scalability and adaptation problem of Unified-MAS in Appendix D.6.

5.2.3 Successful Pattern

To understand this performance leap, we qualitatively compare the nodes generated by Unified-MAS against those from dynamic Automatic-MAS on J1Bench (Appendix E). Dynamic methods like EvoAgent resort to a lazy ensemble approach, generating superficial nodes like "Expert1" and "Expert2" without true domain grounding. In sharp contrast, Unified-MAS synthesizes a highly structured, expert-level judicial pipeline. It explicitly divides reasoning into professional stages: "Legal_Element_Extractor", "Liability_Reasoning", and so on. As detailed in Appendix E, compared to the blind prompt-level voting of original AFlow, the Unified-MAS-enhanced workflow ensures that every stage is traceable and legally grounded.

5.2.4 The Optimization Dynamics

Our reward-based node optimization reveals an important learning dynamic. As shown in Figure 4, the performance trajectory is non-monotonic. From our observation, during early epochs (0 to 5), the system repeatedly targets the most severe "bottleneck node". Updating this node temporarily disrupts established cross-node co-adaptations, caus-

ing short-term perturbation. However, once the bottleneck is sufficiently alleviated, the system shifts focus to other nodes. Consequently, performance rapidly recovers and converges to a sustained global optimum in the later epochs (6–10). These results indicate that our node optimization strategy effectively removes brittle internal logic while avoiding trapping the system in local optima.

6 Conclusion

In this work, we decouple granular node implementation from topology orchestration and propose Unified-MAS, which automatically synthesizes domain-specific nodes through external knowledge retrieval and iteratively refines them via a perplexity-guided reward. Extensive experiments demonstrate that integrating our generated nodes into existing Automatic-MAS approaches universally enhances overall performance, yielding improvements of up to 14.2% while simultaneously reducing costs. Further analysis highlights the robustness of Unified-MAS across different Designer LLMs, demonstrates its generalizability to general domains, and confirms the critical role of the reward-based optimization stage. Moving forward, Unified-MAS can be broadly applied to virtually any specific domain to generate highly professional nodes, seamlessly bridging the gap between general Automatic-MAS and deep domain expertise for future scalable real-world applications.

Limitations

While Unified-MAS demonstrates significant efficacy, we acknowledge certain limitations that present exciting avenues for future research. Primarily, our current framework operates as an offline node-preparation phase, which restricts its immediate applicability in highly dynamic or extremely time-sensitive environments that necessitate real-time, on-the-fly node generation and adaptation. To transition towards fully online, adaptive synthesis, future work should proceed in two main directions. On one hand, future work should focus on streamlining the generation pipeline, allowing the framework to rapidly create and adapt nodes directly. On the other hand, future systems could learn directly from live feedback, quickly adjusting nodes instead of relying on a long offline evaluation.

Acknowledgements

This work is supported by funding from ModelBest.

Ethical Considerations

Unified-MAS reduces the effort required to construct domain-specific multi-agent systems, but generated nodes can inherit factual errors, biases, privacy risks, and correlated failures from their underlying language models. The multi-agent system may produce misinformation, especially when deployed in real-world high-stakes domains such as healthcare, law, finance, or education. We recommend treating it as an assistant system rather than an independent expert, minimizing sensitive data exposure, using least-privilege tool access and human review for consequential outputs.

References

Rahul K Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, and 1 others. 2025. Healthbench: Evaluating large language models towards improved human health. *arXiv preprint arXiv:2505.08775*.

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

Xi Chen, Huahui Yi, Mingke You, WeiZhi Liu, Li Wang, Hairui Li, Xue Zhang, Yingman Guo, Lei Fan, Gang Chen, and 1 others. 2025. Enhancing diagnostic capability with multi-agents conversational large language models. *NPJ digital medicine*, 8(1):159.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2024. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first international conference on machine learning*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. 2025. From llm reasoning to autonomous ai agents: A comprehensive review. *arXiv preprint arXiv:2504.19678*.

Jiayi He, Hehai Lin, Qingyun Wang, Yi R Fung, and Heng Ji. 2025. Self-correction is more than refinement: A learning framework for visual and language reasoning tasks. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 6405–6421.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and 1 others. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55.

Weiquan Huang, Zixuan Wang, Hehai Lin, Sudong Wang, Bo Xu, Qian Li, Beier Zhu, Linyi Yang, and Chengwei Qin. 2026. Ama: Adaptive memory via multi-agent collaboration. *arXiv preprint arXiv:2601.20352*.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38.

Zheng Jia, Shengbin Yue, Wei Chen, Siyuan Wang, Yidong Liu, Zejun Li, Yun Song, and Zhongyu Wei. 2025. Ready jurist one: Benchmarking language agents for legal intelligence in dynamic environments. *arXiv preprint arXiv:2507.04037*.

Prathyusha Jwalapuram, Hehai Lin, Chuyuan Li, Fangkai Jiao, Sudong Wang, Yifei Ming, Zixuan Ke, Chengwei Qin, Giuseppe Carenini, and Shafiq Joty. 2026. The illusion of multi-agent advantage. *arXiv preprint arXiv:2606.13003*.

Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, and 1 others. 2025a. A survey of frontiers in llm reasoning: Inference scaling, learning to reason, and agentic systems. *arXiv preprint arXiv:2504.09037*.

Zixuan Ke, Yifei Ming, Austin Xu, Ryan Chin, Xuan-Phi Nguyen, Prathyusha Jwalapuram, Semih Yavuz, Caiming Xiong, and Shafiq Joty. 2026. Mas-orchestra: Understanding and improving multi-agent reasoning through holistic orchestration and controlled benchmarks. *arXiv preprint arXiv:2601.14652*.

Zixuan Ke, Austin Xu, Yifei Ming, Xuan-Phi Nguyen, Ryan Chin, Caiming Xiong, and Shafiq Joty. 2025b. Mas-zero: Designing multi-agent systems with zero supervision. *arXiv preprint arXiv:2505.14996*.

Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika*, 30(1-2):81–93.

Changlun Li, Yao Shi, Chen Wang, Qiqi Duan, Runke Ruan, Weijie Huang, Haonan Long, Lijun Huang, Nan Tang, and Yuyu Luo. 2025. Time travel is cheating: Going live with deepfund for real-time fund investment benchmarking. *arXiv preprint arXiv:2505.11065*.

Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9.

Hehai Lin, Shilei Cao, Sudong Wang, Haotian Wu, Minzhi Li, Linyi Yang, Juepeng Zheng, and Chengwei Qin. 2025. Interactive learning for llm reasoning. *arXiv preprint arXiv:2509.26306*.

Hehai Lin, Qi Yang, and Chengwei Qin. 2026. Skillmas: Evolving meta-skill for automatic multi-agent systems. *arXiv preprint arXiv:2606.18837*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Junnan Liu, Hongwei Liu, Songyang Zhang, and Kai Chen. 2025b. Rectifying llm thought from lens of optimization. *arXiv preprint arXiv:2512.01925*.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2023. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170*.

MAA-Committees. 2025. Aime problems and solutions.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Bang Liu, Chenglin Wu, Yuyu Luo, and 1 others. 2026. Aorchestra: Automating sub-agent creation for agentic orchestration. *arXiv preprint arXiv:2602.03786*.

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

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, and 1 others. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

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

Kun Wang, Guibin Zhang, ManKit Ye, Xinyu Deng, Dongxia Wang, Xiaobin Hu, Jinyang Guo, Yang Liu, and Yufei Guo. 2025a. Mas ^{2}: Self-generative, self-configuring, self-rectifying multi-agent systems. *arXiv preprint arXiv:2509.24323*.

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

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. 2025b. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*.

Haotian Wu, Shufan Jiang, Mingyu Chen, Yiyang Feng, Hehai Lin, Heqing Zou, Yao Shu, and Chengwei Qin. 2025. Furina: A fully customizable role-playing benchmark via scalable multi-agent collaboration pipeline. *arXiv preprint arXiv:2510.06800*.

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

Tianhan Xu, Ling Chen, Zhe Hu, and Bin Li. 2025. Staf-llm: A scalable and task-adaptive fine-tuning framework for large language models in medical domain. *Expert Systems with Applications*, 281:127582.

Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. 2025. Mas-gpt: Training llms to build llm-based multi-agent systems. *arXiv preprint arXiv:2503.03686*.

Zhe Yu, Wenpeng Xing, Yunzhao Wei, Jie Chen, Hongzhi Wang, Xuyang Teng, and Meng Han. 2026a. Composition collapse: Stable factual knowledge does not imply compositional reasoning. *Preprint*, arXiv:2605.26789.

Zhe Yu, Wenpeng Xing, Tiancheng Zhao, Mohan Li, Changting Lin, and Meng Han. 2026b. Fides: Faithful inference via deep evidence signals for retrieval-memory conflict in rag. *Preprint*, arXiv:2606.05644.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. 2025. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6192–6217.

Cong Zhang, Xin Deik Goh, Dexun Li, Hao Zhang, and Yong Liu. 2025a. Planning with multi-constraints via collaborative language agents. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 10054–10082.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, and 1 others. 2024. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

Yaolun Zhang, Xiaogeng Liu, and Chaowei Xiao. 2025b. Metaagent: Automatically constructing multi-agent systems based on finite state machines. *arXiv preprint arXiv:2507.22606*.

Yubao Zhao, Weiquan Huang, Sudong Wang, Ruochen Zhao, Chen Chen, Yao Shu, and Chengwei Qin. 2026. Training multi-turn search agent via contrastive dynamic branch sampling. *arXiv preprint arXiv:2602.03719*.

A Description of Appendix

The appendix provides extended methodological details and comprehensive experimental data to further support the findings presented in the main manuscript. **Appendix B** presents the detailed pseudocode illustrating the algorithmic workflow of the proposed two-stage Unified-MAS. **Appendix C** provides exhaustive statistics and descriptive summaries of the diverse evaluation benchmarks, detailing dataset splitting protocols and the specific characteristics of each domain-specific task. **Appendix D** delineates the experimental setup, including the baselines, the implementation details, cost analysis, retrieval leakage control, human expert evaluation, discussion, and further ablation studies. **Appendix E** offers a qualitative case study that compares the node generation of Unified-MAS against existing Automatic-MAS. Finally, **Appendix F** catalogs the comprehensive set of prompts utilized for Unified-MAS and our experiment.

B Pseudocode of Unified-MAS

We provide the pseudocode of Unified-MAS here.

Algorithm 1 Unified-MAS

Require: Validation set $\mathcal{D}_{val}$, LLM P_θ, Max epochs K, Balance factor α, Sample size N

Ensure: Domain-specific node set $\mathcal{V}_{domain}$

Stage 1: Search-Based Node Generation

Sample N examples from $\mathcal{D}_{val}$ to form $\mathcal{C}$
Extract keywords across 7 dimensions from $\mathcal{C}$
Synthesize search queries for 4 strategies
Retrieve external knowledge
Generate initial node set $\mathcal{V}_{init} = \{v_1, \ldots, v_m\}$

Stage 2: Reward-Based Node Optimization

$\mathcal{V}_{domain} \leftarrow \mathcal{V}_{init}$
for $k = 1$ to K **do**
  Initialize $R[v] \leftarrow \emptyset$ for all $v \in \mathcal{V}_{domain}$
  for each sample $(q, y) \in \mathcal{D}_{val}$ **do**
    Initialize empty context $A_0 \leftarrow [h_0]$
    Compute baseline predictability: $\mathcal{J}_0 = -\log(\text{PPL}(y|q, A_0))$
    for $t = 1$ to m **do**
      Execute node v_t, obtain reasoning h_t
      Update accumulated context: $A_t \leftarrow [h_0, h_1, \ldots, h_t]$
      Compute: $\mathcal{J}_t = -\log(\text{PPL}(y|q, A_t))$
      Calculate relative gain: $\delta_t = (\mathcal{J}_t - \mathcal{J}_0)/\mathcal{J}_0$
      Compute Improvement Score: $S_{i,t} = \tanh(\delta_t + 1)$
      Compute Consistency Score $\mathcal{S}_{c,t}$ using Eq. (6)
      Node Quality Score: $S_t = (1-\alpha)S_{i,t} + \alpha\mathcal{S}_{c,t}$
      if $t > 1$ **then**
        Node reward: $r_t = S_t - S_{t-1}$
      else
        Node reward: $r_t = S_t$
      end if
      Append r_t to $R[v_t]$
    end for
  end for
  for each node $v \in \mathcal{V}_{domain}$ **do**
    Calculate average reward $\bar{r}(v)$ from $R[v]$
  end for
  Identify $v^* = \arg\min_{v \in \mathcal{V}_{domain}} \bar{r}(v)$
  Retrieve samples where v^* yielded the lowest reward and refine implementation
end for
return $\mathcal{V}_{domain}$

C Statistics of Benchmarks

We split the entire dataset into the validation set and the test set because some Automatic-MAS needs the validation set to sample the best multi-agent system. For fair comparison, all the reported results are based on the test set. We randomly sample some examples from these datasets to build the validation and test set, which can be found in Table 4.

TravelPlanner (Xie et al., 2024): This benchmark aims to evaluate the planning capabilities of language agents within complex, real-world travel scenarios. It features 1,225 meticulously curated user intents, and the evaluation focuses on an agent's proficiency in multi-constraint reasoning and effective tool utilization, serving as a rigorous test for assessing how models navigate intricate planning tasks and integrate disparate information to achieve actionable objectives.

HealthBench (Arora et al., 2025): This benchmark is designed to evaluate the clinical proficiency and safety of AI agents in healthcare. Drawing upon the expertise of 262 practicing physicians across 60 countries, the dataset encompasses 5,000 authentic clinical dialogue scenarios ranging from acute emergencies to global health issues. Utilizing a physician-curated rubric, HealthBench moves beyond simple outcome metrics to rigorously assess models across critical dimensions, including clinical accuracy, communication quality, situational awareness, and safety, thereby ensuring robust performance in high-stakes medical applications.

J1Bench (Jia et al., 2025): This benchmark focuses on automated legal adjudication by simulating court proceedings. The input consists of 93 comprehensive cases, including formal complaints, defendant arguments, and evidentiary materials derived from actual judicial records. The agent is required to synthesize these conflicting testimonies and legal documents to produce a reasoned, final judicial judgment. Evaluation is based on the alignment of the agent's verdict with ground-truth, measuring the model's capacity to interpret legal arguments and arrive at legally sound conclusions.

DeepFund (Li et al., 2025): This benchmark evaluates the financial intelligence of agents in stock market decision-making. The input features a rich, time-sensitive dataset comprising corporate fundamental data, historical price trends, and real-time financial news streams. For a targeted list of stocks, the agent is tasked with outputting a categorical decision, specifically, "Buy", "Sell", or

"Hold". The full dataset contains 139 cases to assess the agent's ability to effectively integrate heterogeneous information into actionable investment strategies.

AIME24&25 (MAA-Committees, 2025): This benchmark collection contains 57 questions and derives from the 2024 and 2025 editions of the American Invitational Mathematics Examination (AIME), comprising two distinct problem sets. Each set contains rigorously vetted mathematical questions characterized by high cognitive demand. The evaluative focus lies in probing advanced mathematical competencies, with particular emphasis on multi-faceted problem-solving strategies that require integration of complex conceptual frameworks.

D Experimental Details

D.1 Specific Manual MAS Baselines

PMC (Zhang et al., 2025a): PMC employs a hierarchical planning framework where a centralized planner decomposes complex tasks into sub-tasks, which are then executed by specialized agents with predefined roles. Incorporating a structured collaboration protocol, it ensures systematic problem-solving across multi-stage reasoning chains.

Diagnosis-MAS (Chen et al., 2025): Diagnosis-MAS utilizes a multi-stage diagnostic workflow where agents engage in iterative feedback loops to identify and mitigate noise in reasoning processes. This approach systematically filters out erroneous information, thereby significantly enhancing the reliability of medical diagnosis.

Court-MAS (Jia et al., 2025): Court-MAS adopts an adversarial interaction model inspired by judicial processes, where agents act as competing parties to present evidence and verify claims. A central judge-agent then adjudicates these contributions based on the simulated interaction.

DeepFund-MAS (Li et al., 2025): DeepFund-MAS implements a multi-agent architecture tailored for financial analysis, where agents are partitioned into functional units such as data acquisition, sentiment analysis, and risk assessment. The system allows agents to correlate disparate financial signals into coherent investment insights.

D.2 Implementation Details

For cost considerations, we set AFlow's maximum number of iterations to 10 and run the validation set once each round. For all other baselines, we strictly follow the original settings. Table 5 lists the important hyperparameters used in Unified-MAS. We set GPT-5-Mini with "low" reasoning effort, while leveraging the standard instruction versions of the other three LLMs. We use GPT-4o as the default LLM-judge following (Ke et al., 2025b).

D.3 Cost Analysis

The node generation and iterative optimization stages are one-time, offline processes. Once the domain-specific node library is constructed, it is repeatedly reused by the Automatic-MAS orchestrator during inference. The inference cost is typically the bottleneck for real-world deployment. Therefore, the efficiency gains we emphasize in the main text and Table 1 refer strictly to the online inference phase. Here we show the offline cost of Unified-MAS's two stages in Table 6. The offline node generation and optimization incur a one-time upfront cost. However, these optimized nodes will reduce the costs required during repeated online inference, leading to a much better long-term performance-cost trade-off.

D.4 Retrieval Leakage Control

We prevent data contamination during the search-based node generation stage in two ways. First, we instruct the LLM to search for domain logic and background knowledge, while explicitly forbidding it from searching for the input question itself. Second, we apply a hard rule-based filter to drop any retrieved documents containing the dataset names. After fetching all the contents, we conduct a rigorous LLM-based inspection on all retrieved documents (TravelPlanner: 289, HealthBench: 202, J1Bench: 266, DeepFund: 248, AIME: 224). The LLM evaluator is prompted to flag any document that: (1) contains the exact question text; (2) contains the direct final answer or official rubric; or (3) contains a paraphrased version preserving the same entities/numbers. The inspection confirms zero instances of contamination.

D.5 Human Expert Evaluation

We invite a finance expert to inspect generated domain-specific nodes and reasoning cases on DeepFund. The expert found that Unified-MAS produces a reasonable investment-committee-style workflow, with multiple specialist agents, adversarial review, conflict resolution, and final decision synthesis. The expert also noted three strengths:

Split	TravelPlanner	HealthBench	J1Bench	DeepFund	AIME24&25
Validation	45	32	16	32	8
Test	180	168	77	107	49

Table 4: Data size for each split in each dataset.

Category	Hyperparameter	Description	Value
Unified-MAS	N	The number of samples used to build context buffer	10
	Turn	The turn number of multi-turn search	10
	α	The weight used to aggregate $\mathcal{S}_{i,t}$ and $\mathcal{S}_{c,t}$	0.6
	K	The epoch number of node optimization stage	10
LLM calls	temperature	The sampling temperature of calling LLM	1.0
	max_tokens	The maximum number of output tokens	32768

Table 5: The description and value of important hyperparameters.

Dataset	Generation	Optimization
TravelPlanner	10.780	4.001
HealthBench	8.033	1.793
J1Bench	11.093	1.737
DeepFund	10.170	3.113
AIME24&25	8.891	0.255

Table 6: Cost (USD $) of Unified-MAS using Gemini-3-Pro as the Designer.

the traces are structured and auditable, the insider-transaction analysis captures important transaction types, and the technical-analysis node produces actionable support/resistance and volume-based signals. However, the multi-agent system may produce misinformation, especially when deployed in real-world high-stakes domains such as healthcare, law, finance, or education. In practical deployment, we recommend that Unified-MAS should be intended to assist domain experts rather than replace human oversight in high-risk decision-making.

D.6 Discussion

At the component level, Unified-MAS involves retrieval, node/prompt synthesis, and validation-based optimization. However, our contribution is to introduce a new mechanism for Automatic-MAS: decoupling domain-specific node implementation from topology orchestration. This is different from standard RAG, prompt optimization, or tool-augmented agents, where the central goal is usually to improve a single agent or a fixed workflow, rather than to relieve an Automatic-MAS orchestrator from the simultaneous design of domain-specific node logic and the search for the global MAS topology.

For scalability, Unified-MAS scales effectively to highly complex domains that require a large number of specialized agents. As task complexity increases, standard Automatic-MAS with pre-defined general nodes typically fail to construct meaningful workflows. Similarly, dynamic generation methods struggle because handling both expert node design and topology orchestration over long horizons overwhelms the LLM. By decoupling node generation, Unified-MAS reliably synthesizes a targeted team of domain experts offline, enabling the orchestrator to scale to complex, long-chain multi-agent tasks.

To solve the real-time adaptation problem in the future, we could introduce a dynamic monitoring mechanism to the framework. For example, if the deployed system encounters an entirely new scenario during online inference (such as a newly enacted financial regulation or a novel medical virus), an "online monitor" could detect the execution failure or low confidence. It could then temporarily trigger a lightweight, real-time version of our search-based generation module to synthesize a new node on the fly, without waiting for a full offline update. Additionally, live execution feedback could be continuously cached to perform rapid, localized node refinement.

D.7 Further Ablation on Hyperparameters

To investigate the effect of some key hyperparameters and setups, we further conduct four ablation studies in Table 7, i.e., Retrieval source, maximum

search turns (Turn), reward weighting factor (α), and size of validation set.

Retrieval Sources: Our three retrieval sources (Google, GitHub, Google Scholar) are complementary and are designed to search different types of external information. Here, we remove the Google Scholar source, and performance noticeably drops, confirming the necessity of combining diverse sources.

Hyperparameters: we test the sensitivity to the maximum search turns (Turn$= 5, 15$), the reward weighting factor ($\alpha = 0.2, 0.4, 0.8$), and validation set size (using only half the set). Our original setting is Full-validation set, Turn$= 10$, and $\alpha = 0.6$. The performance fluctuations are generally minimal across different maximum search turns, reward weighting factors, and validation set sizes. This demonstrates the overall robustness of our framework. For the weighting factor α, the performance remains largely stable around our default setting. We only observe a performance drop in a specific extreme case (i.e., AFlow evaluated with DeepSeek-V3.2 when $\alpha = 0.2$). This occasional variance is expected, as extreme values can disrupt the intended balance between the improvement score and the consistency score. Overall, these results confirm that Unified-MAS is robust and not overly sensitive to hyperparameter choices in most scenarios.

Variants	DeepSeek-V3.2		GPT-5-Mini	
	AFlow	MAS^2	AFlow	MAS^2
Ours	45.79	28.97	44.86	36.45
w/o Google Scholar	40.19	23.36	39.25	30.84
Search Turn = 5	43.93	25.23	41.12	32.71
Search Turn = 15	44.86	28.04	42.06	33.64
$\alpha = 0.2$	39.25	26.17	42.06	32.71
$\alpha = 0.4$	42.06	28.04	45.79	34.58
$\alpha = 0.8$	42.99	24.30	43.93	31.78
Half-validation set	46.73	27.10	42.99	35.51

Table 7: Ablation study results evaluated on DeepFund with Gemini-3-Flash as the Designer.

D.8 More Comparison Experiments

As shown in Table 8, we add more ablation studies on the retrieval source and reward choices, i.e., random bottleneck-node selection, reward-model computation (Dubey et al., 2024), and reward stability under randomized node orders. On one hand, the result shows that all three external sources contribute, while the full Unified-MAS setting performs best. On the other hand, it further suggests that the perplexity-guided reward is effective and reasonably stable: randomized node orders preserve most of the gain, while random bottleneck selection and reward-model computation (Llama-3-8b-rm-mixture) underperform the default design.

Variants	DeepSeek-V3.2
MAS^2	**28.97**
MAS^2 + no retrieval	24.30
MAS^2 + Google only	27.10
MAS^2 + Scholar only	26.16
MAS^2 + GitHub only	25.23
MAS^2 + random bottleneck selection	23.36
MAS^2 + reward-model computation	21.50
MAS^2 + reward stability under randomized orders	27.10

Table 8: More comparison results evaluated on DeepFund with Gemini-3-Flash as the Designer.

In Table 9, we equip MetaAgent, EvoAgent, and AOrchestra with RAG (online searching) as an additional sub-agent, and further evaluate self-consistency with three runs to approximate the offline budget. The MAS^2's results on DeepFund with DeepSeek-V3.2 show that both RAG and self-consistency improve the baselines, but they still remain below Unified-MAS. This suggests that the offline domain-specific node library generated by Unified-MAS provides a stronger improvement than directly attaching RAG to dynamic-generation baselines.

Methods	Original	+ RAG	+ SC=3
EvoAgent	28.97	30.84	36.45
MetaAgent	33.64	35.51	39.25
AOrchestra	25.23	28.97	29.91
Unified-MAS best	**46.73**	-	-

Table 9: Comparison against stronger dynamic Automatic-MAS baselines evaluated on MAS^2, DeepFund with Gemini-3-Flash as the Designer.

E Case Study

Table 10 lists the generated nodes of Unified-MAS using Gemini-3-Pro on TravelPlanner, HealthBench, DeepFund, and AIME24&25. For further comparison, Table 11 shows the generated nodes using Unified-MAS and other Automatic-MAS with dynamic nodes. It indicates that although

these Automatic-MAS can introduce new nodes to some extent, their performance in different specialized fields is not stable enough. For example, EvoAgent generates an excessive number of "Expert Node" to solve the problem in parallel, which is more like an ensemble rather than introducing the real agentic element.

Figure 5 and Figure 6 compare the different MAS generated by MAS-Zero using Gemini-3-Flash, for the same example shown in Table 11, with/without nodes generated by Unified MAS. Compared with the original AFlow, the Unified-MAS version is more structured, transparent, and reliable. It explicitly separates case structuring, legal retrieval, fact verification, damages calculation, and judgment drafting, so each reasoning stage is traceable and easier to validate. By contrast, the original AFlow relies more on prompt-level reasoning and ensemble voting, offering less explicit alignment between evidence, legal rules, and quantified outcomes.

F Prompt Details

We elaborate on the prompts used in Unified-MAS from Figure 7 to Figure 18. These comprehensive instructions cover evaluation and the entire framework pipeline, including keyword extraction, search query generation, strategy analysis, node generation, and node optimization.

The MAS generated by AFlow with Unified-MAS

```
class Workflow:
    def __init__(
        self,
        name: str,
        llm_config,
        dataset: DatasetType,
    ) -> None:
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
        self.case_structurer = operator.CaseStructurer(self.llm)
        self.fact_analyzer = operator.FactAnalyzer(self.llm)
        self.legal_search_engine = operator.LegalSearchEngine(self.llm)
        self.damages_calculator = operator.DamagesCalculator(self.llm)
        self.judgment_drafter = operator.JudgmentDrafter(self.llm)
    async def __call__(self, problem: str):
        """
        Implementation of a structured judicial workflow using specialized operators.
        """
        # Step 1: Extract and structure the key legal elements from raw case text
        s_res = await self.case_structurer(input_data=problem)
        # Prepare inputs for fact analysis (ensuring dictionary types for keys expected to be objects)
        fact_arguments = s_res.get("arguments", {})
        fact_parties = s_res.get("parties", {})

        # Step 2: Retrieve relevant laws and perform forensic fact analysis
        l_res = await self.legal_search_engine(cause_of_action=str(s_res.get("cause_of_action", "")), arguments=str(s_res.get("arguments", "")))
        # Ensure legal search result is a dictionary before accessing "relevant_laws"
        relevant_laws = l_res.get("relevant_laws", str(l_res)) if isinstance(l_res, dict) else str(l_res)
        f_res = await self.fact_analyzer(
            arguments=fact_arguments,
            evidence_summary=str(s_res.get("evidence_summary", "")),
            cause_of_action=str(s_res.get("cause_of_action", "")),
            parties=fact_parties
        )

        # Step 3: Calculate valid damages based on verified facts and legal rules
        c_res = await self.damages_calculator(
            financial_claims=s_res.get("financial_claims", []),
            verified_facts=f_res,
            relevant_laws=relevant_laws,
            cause_of_action=str(s_res.get("cause_of_action", ""))
        )

        # Step 4: Generate a formal draft judgment
        dr_res = await self.judgment_drafter(
            structured_brief=s_res,
            relevant_laws=relevant_laws,
            verified_facts=f_res,
            damages_table=c_res
        )

        # Step 5: Final synthesis and rule enforcement (Statute of Limitations, Divorce grounds, Causality)
        # Safely extract the judgment text from the drafting operator result
        draft_judgment = dr_res.get("final_judgment", str(dr_res)) if isinstance(dr_res, dict) else str(dr_res)
        context = f"Original Case: {problem},
                    Verified Facts: {f_res},
                    Legal Draft: {draft_judgment}"
        solution = await self.custom(input=context, instruction=prompt_custom.JUDGE_PROMPT)
        # Handle final solution format
        if isinstance(solution, dict):
            final_response = solution.get("response", str(solution))
        else:
            final_response = str(solution)
        return final_response, self.llm.get_usage_summary()["total_cost"]
```

Figure 5: The MAS generated by AFlow with Unified-MAS using Gemini-3-Flash as Orchestrator.

The MAS generated by AFlow without Unified-MAS

```
class Workflow:
    def __init__(
        self,
        name: str,
        llm_config,
        dataset: DatasetType,
    ) -> None:
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
        self.sc_ensemble = operator.ScEnsemble(self.llm)
        self.answer_generate = operator.AnswerGenerate(self.llm)
    async def __call__(self, problem: str):
        """
        Implementation of the workflow with Evidence checking, Logical accounting, and Ensemble judgment.
        """
        # Step 1: Objective evaluation of the evidentiary chain and proof status
        evidence = await self.custom(input=problem, instruction=prompt_custom.EVIDENCE_PROMPT)
        evidence_res = evidence.get("response", "")

        # Step 2: Step-by-step logical accounting of liability and proven amounts
        # We use AnswerGenerate to enforce a CoT structure (thought) and a summary (answer)
        logic = await self.answer_generate(input=f"Evidence review result: {evidence_res}, Case information: {problem}, Please strictly deduct amounts not admitted according to the dispute field, then analyze the liability ratio based on the defendant's defense and calculate the final amount.")
        # Safe extraction to prevent KeyError: "answer" if the LLM output doesn't match the expected tags
        logic_thought = logic.get("thought", "No detailed logical analysis")
        logic_answer = logic.get("answer", logic.get("response", "No clear core conclusion"))

        # Step 3: Ensemble multiple candidates to reach the most consistent final judgment
        candidates = []
        for _ in range(3):
            # We pass both the detailed reasoning and the specific conclusion to the judge
            cand_res = await self.custom(
                input=f"Logical analysis: {logic_thought},
                Core conclusion: {logic_answer},
                Case information: {problem}",
                instruction=prompt_custom.JUDGMENT_PROMPT
            )
            candidates.append(cand_res.get("response", ""))
        # Perform self-consistency ensemble to select the most reliable verdict
        solution = await self.sc_ensemble(problem=problem, solutions=candidates)
        return solution.get("response", ""), self.llm.get_usage_summary()["total_cost"]
```

Figure 6: The MAS generated by AFlow without Unified-MAS using Gemini-3-Flash as Orchestrator.

DeepFund		AIME24&25	
Node Name	Node Description	Node Name	Node Description
Fundamental_Analysis_Agent	Analyzes news and policy text for sentiment and key drivers.	Math_Domain_Analyzer	Understands the problem type and key constraints.
Insider_Analysis_Agent	Detects insider trading patterns and conviction signals.	Theorem_Strategy_Retriever	Finds relevant theorems and solving strategies.
Technical_Analysis_Agent	Interprets OHLCV data for trend, support, and resistance levels.	Step_by_Step_Solver	Builds a full solution draft step by step.
Risk_Management_Agent	Checks portfolio cash, exposure, and trading constraints.	Constraint_Logic_Verifier	Checks and fixes logic/math mistakes.
CIO_Synthesis_Agent	Weighs all signals to produce the final trading action.	Final_Answer_Formatter	Extracts and formats the final answer correctly.
HealthBench		**TravelPlanner**	
Node Name	Node Description	Node Name	Node Description
Interaction_Analyzer	Parses conversation history for intent and medical entities.	Requirement_Parser	Extracts dates, budget, party size, and locations from query.
Guideline_Retriever	Retrieves clinical guidelines and red flag protocols via RAG.	Logistics_Planner	Selects flights and hotels within budget constraints.
Clinical_Reasoner	Applies ESI triage logic and generates draft responses.	Daily_Activity_Scheduler	Schedules daily meals and attractions in remaining budget.
Safety_Validator	Verifies safety compliance and hedges diagnostic language.	Final_Formatter	Compiles the plan into the required JSON itinerary format.

Table 10: Unified-MAS's generated nodes using Gemini-3-Pro across four benchmarks.

Prompt for LLM-as-a-judge Evaluation

HealthBench:
Your job is to score a model's reply against a rubric item. Decide whether the reply meets the criterion described in the rubric item.
Model's reply: <reply>
Rubric item: <rubric_item>
Instructions
Return a json object with the following fields: "explanation" and "criteria_met".
The "explanation" field should be a string explaining why the reply does or does not meet the criteria of the rubric item.
The "criteria_met" field should be a boolean indicating whether the reply meets the criteria of the rubric item. If a rubric item has multiple sentences or criteria, you should consider all of them. If any of the criteria is not met, the answer should be false. Only return true if all of the criteria are met.
One important exception: if a criteria says "such as", "for example", or "including", the reply does not have to include all of the examples listed to meet the criteria.
Final instruction
Return just the json object in markdown format. Do not include any other text in the response.

J1Bench:
You are a legal expert. Please determine whether the "Answer to be evaluated" fully and accurately covers all the core points of the "Standard answer" based on the "legal issue" and the "Standard Answer", and then give a score ranging from 0 to 10. You don't need to consider non-substantive factors such as whether the answer to be evaluated is expressed concisely, whether the key points are highlighted, whether small talk is used, or whether the structure is lengthy. You don't need to deduct points for being insufficiently concise, nor do you need to consider the difference in language forms between Chinese and English. You only need to consider whether the content meaning is consistent.
Standard answer: <gt_answer>
Answer to be evaluated: <mode_answer>
Output your results in the following format (no line breaks, no parentheses): <Rating: ..., Reason: ...>

Figure 7: Prompt for LLM-as-a-judge Evaluation.

Method	Node Name	Node Description / Function
Input Question: You are a rigorous and impartial presiding judge. Your task is to generate legal reasoning and deliver the final ruling based on the plaintiff's and defendant's statements and the additional information provided. Maintain a neutral, professional, and fair judicial tone at all times, without favoring either side. You are given the following information: `{"category": "Personality rights dispute", "plaintiff": "xx Song", "defendant": "A kindergarten in Beijing", "incident": "2023-04-21 kite activity injury (facial cut near eye)", "claims":[medical 6900¥, lost wages 4000¥, transport 3000¥, mental distress 10000¥, future treatment 40000¥], ...}`		
AOrchestra (Gemini-3-Flash)	`MainAgent`	Top-level orchestrator deciding next action.
	`error`	Runtime error transition captured in trajectory log.
	`delegate_task`	Delegates current sub-problem to a sub-agent.
	`finish`	Sub-agent final answer step for delegated task.
	`complete`	MainAgent composes and returns final answer.
	`SubAgent`	Delegated worker agent that executes subtask reasoning.
EvoAgent (Gemini-3-Flash)	`MainAgent`	Controls iterative expert evolution and selection.
	`Expert#1`	Expert role (tort-law doctrine and social public policy).
	`Expert#2`	Expert role (protection of minors' rights and mental-health assessment).
	`Expert#3`	Expert role for refining disputed issues (future treatment and long-term impact).
	`ExpertGroup(3)`	Aggregated 3-expert panel output per iteration.
MetaAgent (Gemini-3-Flash)	`Presiding_Judge`	Performs legal analysis: statute search, liability split, claim acceptance/rejection, and summary for downstream actuarial calculation.
Unified_MAS (Gemini-3-Flash)	`Rhetorical_Segmenter`	Segments legal input into modules: plaintiff claims, defense, findings, and evidence.
	`Legal_Element_Extractor`	Extracts legal-technical elements such as claim items, amounts, and injury/contract details.
	`Statutory_Retriever`	Retrieves applicable PRC Civil Code statutes based on extracted legal elements.
	`Evidence_Evaluator`	Evaluates evidentiary support using civil "high probability" proof standard.
	`Liability_Reasoning_Engine`	Applies law to verified facts to infer liability ratio and compensation basis.
	`Final_Judgement_Synthesizer`	Produces final judicial reasoning and verdict in required output format.
Unified_MAS (GPT-5-Mini)	`Ingest_and_Normalize`	Normalizes input into canonical text blocks with metadata and offsets.
	`Document_Classifier`	Classifies document/domain type and extracts top remedies.
	`Party_and_Role_Extraction`	Extracts parties/roles with provenance.
	`Claims_and_Remedies_Extraction`	Extracts requested claims/remedies and maps them to claimants.
	`Evidence_Enumeration`	Enumerates/classifies evidence and links evidence to claims/events.
	`Timeline_and_Causation`	Builds a chronological event timeline and causal links to damages.
	`Retrieve_Statutes_and_Precedents`	Retrieves legal statutes and precedent snippets (RAG).
	`Statute_to_Fact_Linking`	Links facts/claims to statute or case references with justifications.
	`Liability_Reasoning`	Infers party liability allocation with legal rationale.
	`Damage_Calculation_and_Reconciliation`	Performs component-level damage calculation and reconciliation.
	`Validation_and_Consistency_Checks`	Runs consistency/constraint checks on full structured output.
	`Final_Judgment_Synthesis`	Synthesizes full Chinese judgment text and structured verdict.
	`Final_Answer_Line`	Emits the final one-line verdict beginning with "Answer:".
Unified_MAS (Gemini-3-Pro)	`Case_Structurer`	Parses raw case JSON into parties, cause of action, claims, and dispute summary.
	`Legal_Search_Engine`	Retrieves statutes/judicial interpretations relevant to the dispute type.
	`Fact_Analyzer`	Verifies facts and causality from conflicting statements and evidence.
	`Damages_Calculator`	Validates and computes monetary compensation items.
	`Judgment_Drafter`	Drafts the final formal judgment text from structured reasoning.

Table 11: Comparison of generated nodes using MetaAgent, EvoAgent, AOrchestra, and Unified-MAS on J1Bench.

Prompt for Keyword Extraction

System_prompt:
You are an expert dataset and task analyst. You are given multiple samples from a benchmark dataset. Your task is to carefully read all samples, analyze this Specific Domain Task, and extract keywords across six specific dimensions required to solve this task. The extracted keywords should be concise and representative, and should not focus on the specific data samples, but on the general domain and task.
User_prompt:
User Task samples: <samples_text>
Analyze the description above and reason to extract keywords for the following six dimensions. For each dimension, provide 5-10 most representative terms:
1. **Domain:** The macro industry background (e.g., Fintech, Supply Chain, Bioinformatics, etc.).
2. **Task:** The core technical problem to solve (e.g., Anomaly Detection, Named Entity Recognition, Summarization, etc.).
3. **Entities:** The specific data objects or physical entities involved (e.g., Transaction Logs, PDF Contracts, Protein Sequences, Sensor Data, etc.).
4. **Actions:** The specific operations performed on the data (e.g., Classify, Extract, Reason, Optimize, Verify, etc.).
5. **Constraints:** Performance metrics or limitations (e.g., Low Latency, Privacy Preserving, Explainability, Offline Inference, etc.).
6. **Desired Outcomes:** The expected results or metrics (e.g., Accuracy, Precision, Recall, F1 Score, AUC, MAP, NDCG, etc.).
7. **Implicit Knowledge:** Based on your expert knowledge, infer specific jargon, SOTA techniques, common challenges, or potential risks that are not explicitly mentioned but are essential for solving this problem (e.g., "Imbalanced Data" for fraud, "Hallucination" for GenAI, "Bullwhip Effect" for supply chain, etc.).
 # Output Format
Please output both your thinking and answer in the JSON format.
"thinking" entry: [Your thinking process, how you arrive your answer]
"answer" entry: [your answer in the JSON format]
For the "thinking" entry, you need to first carefully read the <samples_text>, summarize the task description, and then reason step by step to arrive at your answer.
For the "answer" entry, please output a valid JSON object. Do not include any conversational filler or markdown formatting outside the JSON code block. Format as follows:

```
{{
    "Domain": ["..."],
    "Task": ["..."],
    "Entities": ["..."],
    "Actions": ["..."],
    "Constraints": ["..."],
    "Desired_Outcomes": ["..."],
    "Implicit_Knowledge": ["..."]
}}
```

Figure 8: Prompt for Keyword Extraction.

Prompt for Search Query Generation

System_prompt:
You are an expert in Information Retrieval (IR) and Multi-Agent System Design. You know how to construct precise search queries to retrieve background knowledge, high-quality academic papers, code implementations, and industry Standard Operating Procedures (SOPs).
User_prompt:
Based on the provided [Structured Keywords (Domain, Task, Entities, Actions, Constraints, Desired_Outcomes, Implicit_Knowledge)], apply four specific search strategies to generate a list of search queries for Google Scholar, GitHub, and General Web Search.
Structured Keywords JSON: <keywords_json_str>
Apply the following four strategies to construct your queries for each dimension:
1. **Strategy A: Background Knowledge**
Logic: Domain + Implicit_Knowledge
Aim: Use domain jargon to find background knowledge, cutting-edge solutions, theoretical frameworks, and surveys.
2. **Strategy B: High-quality Academic Papers about System Architecture (Workflow & Design)**
Logic: Task + Constraints
Aim: Find architectural designs (e.g., Router, Pipeline, Map-Reduce) that satisfy specific constraints (e.g., Privacy, Real-time).
3. **Strategy C: Technical Code Implementation**
Logic: Entities + Actions
Aim: Find code repositories, libraries, or preprocessing tools for specific data types.
4. **Strategy D: Evaluation & Metrics**
Logic: Task + Desired_Outcomes
Aim: Find standard datasets and quantitative metrics to evaluate the Agent's performance.
 # Output Instructions
Generate 5-10 search queries for EACH strategy. Use Boolean operators (AND, OR) where appropriate to optimize results.
Please output ONLY a valid JSON object with the following structure:

```
{{
    "strategy_A": [
        {{"query": "...", "reasoning": "Using [Implicit Term] to find advanced patterns"}}
    ],
    "strategy_B": [
        {{"query": "...", "reasoning": "To find architectures satisfying [Constraint]"}}
    ],
    "strategy_C": [
        {{"query": "...", "reasoning": "To find tools for processing [Object] via [Action]"}}
    ],
    "strategy_D": [
        {{"query": "...", "reasoning": "To find benchmarks for [Outcome]"}}
    ]
}}
```

Figure 9: Prompt for Search Query Generation.

Epoch	Node Internal Implementation

Epoch 0 (Initialization)	`def Fact_Analyzer(self, input_data):`
	`  args = input_data.get("arguments", {})`
	`  evid = input_data.get("evidence_summary", {})`
	`  node_messages = [`
	`  {"role":"system","content":"You are a Senior Investigator..."},`
	`  {"role":"user","content":f"Plaintiff's Story & Evidence: {args}, {evid}..."}`
	`  ]`
	`  response = self.llm_client.chat(node_messages, response_format="json_object")`
	`  return response`

Epoch 10 (Optimized)	`def Fact_Analyzer(self, input_data):`
	`  # omitted detailed prompt bodies for space`
	`  cause_of_action = input_data.get('cause_of_action', 'Civil Dispute')`
	`  parties = input_data.get('parties') or input_data.get('specific_characters', {})`
	`  arguments = input_data.get('arguments', {})`
	`  evidence = input_data.get('evidence', {})`
	`  evidence_str = json.dumps(evidence, ensure_ascii=False, indent=2)`
	`  fact_finder_messages = [system: category-aware rules for Tort/Financial/Divorce, user: case context]`
	`  fact_analysis = self.llm_client.chat(fact_finder_messages)`
	`  judge_messages = [system: liability matrix, user: Fact Analysis]`
	`  response = self.llm_client.chat(judge_messages, response_format='json_object')`
	`  return response`

Table 12: Comparison of `Fact_Analyzer` implementation across epochs on J1Bench. Compared to the unoptimized Epoch 0, Epoch 10 contains a two-stage, category-aware reasoning pipeline.

Prompt for Multi-turn Search

System_prompt: You are a web search controller. Your job is to decide, step by step, how to search the web so that the user can find content that matches the target description. At each round, you will see the target description and a summary of past searches and results, and you must decide whether more searching is needed. You MUST respond with a valid JSON object only, no extra text.
User_prompt:
Target description: <target_description>
Search round: <round_idx> / <max_rounds>
Past search rounds: <history_str>
Search engine context:
Backend type: <engine_type>
Instructions:
Github_engine_hint:
Current search backend: GitHub repository search. You MUST construct queries that look like GitHub repo searches, NOT natural language questions. Focus on a few core keywords: domain, task, entities, and techniques. Prefer short keyword-style queries, optionally with GitHub qualifiers such as 'language:python', 'in:name,description,readme', 'stars:>10'. Avoid 'survey of', 'methods for', 'towards', or very long sentences in the query.
Scholar_engine_hint:
Current search backend: Google Scholar (academic papers). You should construct queries that look like paper titles or combinations of technical terms. It is good to include phrases like 'survey', 'review', 'state of the art' when searching for overviews. Focus on scientific keywords (task, domain, methodology) rather than implementation details.
Google_engine_hint:
Current search backend: general Google web search. You may mix natural language with key technical terms. Focus on retrieving background knowledge, blog posts, documentation, or tutorials relevant to the target description.
Your task in THIS round:
1. Carefully read the target description and past search results.
2. Decide whether we already have enough information that clearly matches the target description.
3. If yes, set "done": true and summarize the useful information we already have.
4. If no, set "done": false and propose the NEXT web search query to run.
Output JSON schema (you must strictly follow):
{{
"done": bool, // true if we already have enough matching information
"need_search": bool, // whether to run another web search in this round
"next_query": str, // the next search query to run (empty if done=true)
"reasoning": str, // your reasoning for this decision
"summary": str // if done=true, summarize what has been found and why it matches
}}

Figure 10: Prompt for Multi-turn Search.

Prompt for Strategy_A Analysis

System_prompt:
You are an expert technical analyst. Your task is to analyze multiple documents (PDFs and TXTs) that were retrieved through a web search for background knowledge related to a specific task, and provide a comprehensive summary.
User_prompt:
Task Description (from task_keywords thinking) <task_thinking>
Strategy: <strategy_name>
Documents Retrieved: <files_text>
IMPORTANT:
Please provide an EXTREMELY DETAILED and COMPREHENSIVE analysis. The more detailed, the better. Include specific examples, step-by-step explanations, concrete details, and thorough descriptions.
Your task:
1. Analyze all the documents above and identify which aspects/aspects they discuss the background knowledge related to the task described above. Be very specific and detailed about each aspect.
2. Summarize the key background information that is needed to solve this task. Provide EXTREMELY DETAILED descriptions, including but not limited to:

Overall task workflow and processes: Provide a DETAILED, step-by-step workflow with specific stages, decision points, inputs/outputs at each stage, and the complete process flow. Include concrete examples and detailed explanations of each step.

Key points and important considerations: List ALL important points with detailed explanations, why they matter, and how they impact the task. Be thorough and comprehensive.

Domain-specific knowledge and terminology: Provide detailed definitions, explanations, and context for each term. Include how these concepts relate to each other and their significance in the domain.

Relevant frameworks, methodologies, or approaches: Describe each framework/methodology in DETAIL, including their components, how they work, when to use them, and their advantages/disadvantages. Provide specific examples.

Common challenges and solutions: Detail each challenge with specific scenarios, root causes, and provide detailed solutions with step-by-step approaches. Include real-world examples.

Best practices and standards: Provide detailed best practices with specific guidelines, checklists, and detailed explanations of why each practice is important.

3. Provide a structured summary that clearly explains:

What background knowledge aspects are covered in these documents (with detailed descriptions)
What specific background information is needed to solve the task (be very specific and detailed)
How this background knowledge relates to the task at hand (provide detailed connections and relationships)

Remember:
The more detailed and comprehensive your analysis, the better. Include specific examples, detailed explanations, step-by-step processes, and thorough descriptions throughout.
Please provide a comprehensive and well-structured analysis in JSON format:

```
{{
    "aspects_covered": ["detailed aspect1 with explanation", "detailed aspect2 with explanation", ...],
    "background_information": {{
        "task_workflow": "DETAILED step-by-step workflow with all stages, inputs/outputs, decision points, and complete process flow. Be extremely thorough.",
        "key_points": ["detailed point1 with full explanation", "detailed point2 with full explanation", ...],
        "domain_knowledge": "DETAILED explanation of domain-specific knowledge, terminology, concepts, and their relationships. Be comprehensive and thorough.",
        "frameworks_methodologies": ["detailed framework1 with components and usage", "detailed framework2 with components and usage", ...],
        "challenges_solutions": "DETAILED description of common challenges with specific scenarios, root causes, and detailed step-by-step solutions with examples.",
        "best_practices": "DETAILED best practices with specific guidelines, checklists, and explanations of importance. Be comprehensive."
    }},
    "summary": "EXTREMELY DETAILED and comprehensive summary of the background knowledge, including all key points, detailed workflows, and thorough explanations..."
}}
```

Figure 11: Prompt for Strategy_A Analysis.

Prompt for Strategy_B Analysis

System_prompt:
You are an expert system architect and technical analyst. Your task is to analyze academic papers and documents about system architecture, workflow, and design related to a specific task, and provide insights on architectural patterns and design approaches.
User_prompt:
Task Description (from task_keywords thinking): <task_thinking>
Strategy: <strategy_name>
Documents Retrieved: <files_text>
IMPORTANT:
Please provide an EXTREMELY DETAILED and COMPREHENSIVE analysis. The more detailed, the better. Include specific architectural diagrams, descriptions, detailed workflow steps, component interactions, and thorough explanations.
Your task:
1. Analyze all the documents above and identify the system architectures, workflows, and design patterns they discuss. Be very specific and detailed about each pattern and architecture.
2. Summarize the key architectural and design information relevant to solving this task. Provide EXTREMELY DETAILED descriptions, including but not limited to:
System architecture patterns and structures: Provide DETAILED descriptions of each architecture pattern, including components, their roles, data flow, communication patterns, and how they work together. Include specific examples and detailed explanations.
Workflow designs and process flows: Provide EXTREMELY DETAILED, step-by-step workflow descriptions with all stages, transitions, decision points, data flows, error handling, and complete process flows. Include detailed diagrams, descriptions, and specific examples.
Component interactions and interfaces: Detail how components interact, what interfaces they use, data formats, protocols, and communication mechanisms. Be very specific and thorough.
Design principles and constraints: Provide detailed explanations of each design principle (e.g., privacy, real-time, scalability) with specific implementation strategies, trade-offs, and detailed guidelines. Include concrete examples.
Architectural trade-offs and decisions: Detail each trade-off with specific scenarios, pros/cons, decision criteria, and detailed explanations of why certain choices are made. Be comprehensive.
Best practices for system design: Provide detailed best practices with specific guidelines, patterns to follow, anti-patterns to avoid, and detailed explanations. Include real-world examples.
3. Provide a structured summary that clearly explains:
What architectural patterns and workflows are covered in these documents (with detailed descriptions)
What specific architectural/design information is needed to solve the task (be very specific and detailed)
How these architectural approaches relate to the task requirements (provide detailed connections and relationships)
Remember:
The more detailed and comprehensive your analysis, the better. Include specific architectural details, detailed workflow steps, component interactions, and thorough explanations throughout.
Please provide a comprehensive and well-structured analysis in JSON format:
{{
"architectural_patterns": ["detailed pattern1 with components and structure", "detailed pattern2 with components and structure", ...],
"design_information": {{
"system_architectures": "DETAILED description of system architectures with components, data flows, communication patterns, and how they work together. Be extremely thorough.",
"workflow_designs": ["DETAILED step-by-step workflow1 with all stages and transitions", "DETAILED step-by-step workflow2 with all stages and transitions", ...],
"component_interactions": "DETAILED description of component interactions, interfaces, data formats, protocols, and communication mechanisms. Be comprehensive.",
"design_constraints": ["detailed constraint1 with implementation strategies", "detailed constraint2 with implementation strategies", ...],
"architectural_tradeoffs": "DETAILED description of trade-offs with specific scenarios, pros/cons, decision criteria, and explanations. Be thorough.",
"design_best_practices": "DETAILED best practices with specific guidelines, patterns, anti-patterns, and explanations. Include examples. Be comprehensive."
}},
"summary": "EXTREMELY DETAILED and comprehensive summary of the architectural and design knowledge, including all patterns, detailed workflows, and thorough explanations..."
}}

Figure 12: Prompt for Strategy_B Analysis.

Prompt for Strategy_C Analysis

System_prompt:
You are an expert AI system architect and LLM prompt engineer. Your task is to analyze code repositories and design frameworks for solving tasks using Large Language Models (LLMs). Focus on high-level architecture, operation design, and how to migrate traditional ML/small model approaches to LLM-based solutions.
User_prompt:
Task Description (from task_keywords thinking): <task_thinking>
Strategy: <strategy_name>
Documents Retrieved (Code Repositories): <files_text>
IMPORTANT: Focus on FRAMEWORK DESIGN and LLM MIGRATION, not on specific libraries or dependencies. Think about how to solve the task at a high level using LLMs.
Your task:
1. Analyze the overall framework and architecture in the provided code:
What is the high-level workflow and operation flow? How are different components organized and connected? What are the key operations/steps needed to solve the task? How can these operations be efficiently designed and orchestrated?
2. Design LLM-based solutions to replace or enhance the small model implementations:
Operation Design: How to break down the task into well-defined operations that can be executed by LLMs? What operations are needed and how should they be structured? **Prompt Engineering:** For each operation that was previously done by small models, design detailed prompts for LLMs. What should be the input format, what instructions should be given, and what output format is expected? **Model-level Mechanisms:** How to implement global constraint checking, validation, error handling, and other model-level controls? What mechanisms are needed to ensure the LLM operations work correctly together? **Data Flow:** What is the input/output format for each LLM operation? How should data flow between different operations? What transformations are needed?
3. Migration Strategy:
How can the existing small model code be adapted to use LLMs instead? What are the key differences in approach between small models and LLMs for this task? How to design the system to leverage LLM capabilities while maintaining the original workflow structure?
4. Framework Considerations:
What is the overall system architecture needed to solve this task? How should operations be orchestrated and sequenced? What are the critical decision points and branching logic? How to handle state management and context passing between operations?
Focus Areas (in order of importance):
(1) Overall Framework & Architecture: How to structure the solution at a high level. (2) Operation Design: How to break down the task into LLM-executable operations. (3) Prompt Design: Detailed prompt templates for each LLM operation. (4) Data Processing & Flow: Input/output formats and data transformations between operations. (5) Model-level Mechanisms: Global constraints, validation, error handling. (6) Migration Strategy: How to adapt small model code to LLM-based approach.
Do NOT focus on: Specific library dependencies or installation requirements. Environment setup details. Low-level implementation details of non-LLM components.
Please provide a comprehensive and well-structured analysis in JSON format:
{{
"overall_framework": {{ "architecture": "DETAILED description of the overall system architecture and framework design needed to solve this task. Explain the high-level structure, component organization, and how different parts work together.", "workflow": "DETAILED step-by-step workflow description. Explain the sequence of operations, decision points, and how the system processes the task from start to finish.", key_operations": ["operation1: detailed description of what it does and how it fits in the framework", "operation2: ...", ...] }},
"llm_migration": {{ "operation_design": "DETAILED description of how to design operations for LLM execution. Explain how to break down the task into operations, how operations should be structured, and how they should interact.", "prompt_templates": [{{ "operation_name": "name of the operation", "purpose": "what this operation does in the overall framework", "input_format": "detailed description of input format and structure", "prompt_template": "detailed prompt template with placeholders and instructions", "output_format": "detailed description of expected output format", "constraints": "any constraints or validation rules for this operation" }}, ...], "model_level_mechanisms": "DETAILED description of model-level mechanisms needed: global constraint checking, validation rules, error handling strategies, state management, context passing, etc. Be very specific about how these mechanisms work.", "migration_strategy": "DETAILED explanation of how to migrate from small model code to LLM-based approach. What changes are needed, what can be reused, and how to adapt the existing workflow." }},
"data_processing": {{ "input_output_formats": "DETAILED description of input/output formats for LLM operations. What data structures are needed, what format should be used, and how data should be structured.", "data_flow": "DETAILED description of how data flows between operations. What transformations are needed, how to pass context between operations, and how to maintain data consistency.", "preprocessing": "DETAILED description of any preprocessing needed before sending data to LLMs (if any).", "postprocessing": "DETAILED description of any postprocessing needed after receiving LLM outputs (if any)." }},
"summary": "EXTREMELY DETAILED and comprehensive summary of the framework design, operation structure, LLM migration strategy, and how to solve this task using LLMs. Include specific examples of prompt designs, operation flows, and architectural decisions."
}}

Figure 13: Prompt for Strategy_C Analysis.

Prompt for Strategy_D Analysis

System_prompt:
You are an expert evaluator and metrics analyst. Your task is to analyze documents about evaluation metrics, benchmarks, and assessment methods related to a specific task, and provide insights on evaluation approaches and standards.
User_prompt:
Task Description (from task_keywords thinking): <task_thinking>
Strategy: <strategy_name>
Documents Retrieved: <files_text>
IMPORTANT:
Please provide EXTREMELY DETAILED and COMPREHENSIVE analysis. The more detailed, the better. Include specific metric definitions, detailed evaluation procedures, step-by-step assessment workflows, and thorough explanations.
Your task:
1. Analyze all the documents above and identify the evaluation metrics, benchmarks, and assessment methods they discuss. Be very specific and detailed about each metric and method.
2. Summarize the key evaluation information relevant to solving this task. Provide EXTREMELY DETAILED descriptions, including but not limited to:
Standard evaluation metrics and their definitions: Provide DETAILED definitions for each metric, including mathematical formulas, calculation methods, interpretation guidelines, and specific use cases. Include examples and detailed explanations.
Benchmark datasets and evaluation protocols: Detail each dataset with size, format, structure, quality, and provide DETAILED evaluation protocols with step-by-step procedures, data splits, evaluation criteria, and complete assessment workflows. Be extremely thorough.
Assessment methodologies and procedures: Provide DETAILED, step-by-step assessment workflows with all stages, evaluation criteria, scoring methods, and complete procedures. Include specific examples and detailed explanations.
Performance standards and baselines: Detail performance benchmarks with specific numbers, comparison methods, baseline implementations, and detailed explanations of what constitutes good performance. Be comprehensive.
Evaluation best practices and guidelines: Provide detailed best practices with specific guidelines, common mistakes to avoid, validation procedures, and detailed explanations. Include real-world examples.
Metrics interpretation and analysis methods: Detail how to interpret each metric, what values indicate good/bad performance, statistical analysis methods, and detailed interpretation guidelines. Be thorough.
3. Provide a structured summary that clearly explains:
What evaluation metrics and benchmarks are covered in these documents (with detailed descriptions)
What specific evaluation information is needed to assess task performance (be very specific and detailed)
How these evaluation approaches relate to the task requirements (provide detailed connections and relationships)
Remember:
The more detailed and comprehensive your analysis, the better. Include specific metric definitions, detailed evaluation procedures, step-by-step workflows, and thorough explanations throughout.
Please provide a comprehensive and well-structured analysis in JSON format:
{{
"evaluation_metrics": ["detailed metric1 with definition and formula", "detailed metric2 with definition and formula", ...],
"evaluation_information": {{
"standard_metrics": ["detailed metric1 with calculation method", "detailed metric2 with calculation method", ...],
"benchmark_datasets": ["detailed dataset1 with protocol", "detailed dataset2 with protocol", ...],
"assessment_methodologies": "DETAILED step-by-step assessment workflow with all stages, criteria, scoring methods, and complete procedures. Be extremely thorough.",
"performance_standards": "DETAILED performance benchmarks with specific numbers, comparison methods, baselines, and explanations. Be comprehensive.",
"evaluation_best_practices": "DETAILED best practices with guidelines, common mistakes, validation procedures, and explanations. Include examples. Be comprehensive.",
"metrics_interpretation": "DETAILED interpretation guidelines with analysis methods, value meanings, and statistical considerations. Be thorough."
}},
"summary": "EXTREMELY DETAILED and comprehensive summary of the evaluation and metrics knowledge, including all metrics, detailed procedures, and thorough explanations..."
}}

Figure 14: Prompt for Strategy_D Analysis.

Prompt for Node Template

```
def {node_name}(self, input_data):
"""
node_id: {node_id}
node_type: {node_type}
description: {description}
dependencies: {dependencies}
input: {input}
output: {output}
"""
# ———- Step 1: Process the input data
# input_data is a dictionary with the keys as the input names and the values as the input values
# First, extract the input values from the input_data dictionary
# Fill your code here


# ———- Step 2: Implement the node logic for one of the node types (LLM_Generator, Retrieval_RAG)
# Second, for LLM_Generator nodes, use LLMs to process the input data
# For example, define the system prompt and user prompt:
# node_messages = [
# {"role": "system", "content": System Prompt from the prompt_template},
# {"role": "user", "content": User Prompt from the prompt_template (embed the input values) + Constraints from the constraints field},
# ]
# Then, call the LLM to get the output. If there are multiple LLM calls, you should call the LLMs according to the logic_description field.
# For example, use self.llm_client.chat(node_messages, response_format='json_object') to get the json format output
# Use self.llm_client.chat(node_messages, response_format='normal') to get the normal text output
# Fill your code here


# For Retrieval_RAG nodes, find the information that this node needs to retrieve from the logic_description
# Use self.search_engine.multi_turn_search(information needed to retrieve) to get the retrieved context
# Based on the retrieved context, use the summarization prompt template (marked as User Prompt: in the prompt_template) to summarize the retrieved context
# Use self.llm_client.chat(node_messages, response_format='json_object') to get the json format output
# Use self.llm_client.chat(node_messages, response_format='normal') to get the normal text output
# Fill your code here


# ———- Step 3: Collect the output
# Finally, collect the output into a dictionary with the keys as the output names and the values as the output values
# Fill your code here


return output_data
```

Figure 15: Prompt for Node Template.

Prompt for Node Generation Part 1

System_prompt:
You are an expert system architect and multi-agent system designer. Your task is to design a complete pipeline of nodes (operators) to solve a specific task based on the task description and strategy analysis. You must carefully identify every step the task requires and create a corresponding node for each, do not omit necessary steps. You may ONLY use two types of nodes: **LLM_Generator** (call LLM to do reasoning/generation) and **Retrieval_RAG** (use search engine for RAG). All verification, validation, parsing, and format-checking must be implemented via the LLM (by writing clear requirements and rules in the prompt_template so the LLM performs checks and outputs the correct format). Do NOT write code to verify, parse, or validate LLM outputs, use the LLM to do it. Each node must follow the provided node definition structure and work together to form a complete solution pipeline.
User_prompt:
Task Description: <task_thinking>
<task_samples_section>
Strategy Analysis: <strategy_analysis>
The code template for all nodes is (Only use for the all_code field in the node definition): <code_template>
IMPORTANT: Design a pipeline using ONLY two node types. Each node must follow the node definition structure above.
STRICT RULES:
Allowed node types: LLM_Generator and Retrieval_RAG.
Verification and parsing via LLM, NOT code: Any need for verification (e.g. format check, validity check, number validation), parsing (e.g. extracting structured data from text), or fixing malformed output must be implemented by the LLM: put the rules and expected output format in the prompt_template (System Prompt / User Prompt) so that the LLM performs the checks and returns well-formed output. Do NOT write Python code to validate (e.g. json.loads, try/except, re.match) or parse LLM responses, if output might be messy, add instructions in the prompt or add another LLM_Generator node that asks the LLM to clean/validate and re-output.
For calculations or deterministic steps, use an LLM_Generator node: ask the LLM to perform the reasoning and output the result in the required format; do not use code.
Your task:
1. Analyze the task and strategy analysis to understand: What is the overall task that needs to be solved? What background knowledge, architectural patterns, and evaluation metrics are available? List exhaustively all operations and workflow steps the task requires (e.g. input parsing, fact extraction, knowledge retrieval, reasoning, validation, synthesis, final answer formatting). Do not skip or merge steps mentally, write them down. Each of these should eventually map to at least one node.
2. Design a complete pipeline of nodes using ONLY LLM_Generator and Retrieval_RAG:
For each step you identified above, create a corresponding node. Do not generate too few nodes: the pipeline must have enough nodes to cover the entire task from input to final output. If the task typically needs e.g. extraction → retrieval → reasoning → synthesis → formatting, you must have nodes for each (or clearly combined in a justified way). Break down the task into logical steps; each step is either (a) call LLM to do something, or (b) use search engine to retrieve then LLM to summarize/use.
Before finalizing the node list, double-check: Is there a node that handles retrieval if the task needs external knowledge? Is there a node that produces the final answer in the required format? Are there nodes for every distinct logical phase (e.g. understand input, gather context, reason, output)? Add nodes if any required step is missing.
Nodes are connected through dependencies (dependencies field).
Do NOT add any node that would require custom Python code (e.g. no "Calculator Tool", "Validator Tool", "Parser Tool" as Python code). Use LLM_Generator for such roles if needed.
3. For each node, provide complete information following the node definition:
node_name: A descriptive name (e.g., "xx_Agent").
node_type: One of [LLM_Generator, Retrieval_RAG].
description: Summary of the node's role in the pipeline.
dependencies: List of upstream node names that this node depends on.
input: What information this node reads from inputs (be specific based on task samples).
output: What this node produces (be specific about output format).
constraints: Global constraints this node must comply with (from task requirements).
implementation: logic_description: Detailed description of the implementation logic (no code; describe what the node does in terms of LLM calls and/or search + LLM). prompt_template: (For both node types) MUST provide complete, detailed prompt content: System Prompt (marked as "System Prompt:") and User Prompt (marked as "User Prompt:") with placeholders. Be specific and include examples. tools_needed: For Retrieval_RAG nodes use ["Search"]; for LLM_Generator use []. Do NOT include "code_snippet". Omit it or set to null.
all_code: Minimal runnable code only: (1) Read inputs from input_data. (2) For LLM_Generator: fill the prompt_template with input values and call self.llm_client.chat(node_messages, response_format=...). (3) For Retrieval_RAG: build search query from inputs, call self.search_engine.multi_turn_search(query), then fill prompt_template with retrieved context and call self.llm_client.chat. (4) Return output_data dict. Do NOT add code that verifies, parses, or validates the LLM response (no json.loads, re, try/except for parsing, no format checks)—all verification/parsing is done by the LLM via the prompt.

Figure 16: Prompt for Node Generation Part 1.

Prompt for Node Generation Part 2

CRITICAL:

Verification and parsing LLM's job: If a node needs to ensure valid JSON, correct format, or validated numbers, write these requirements in the prompt_template (e.g. "Output only valid JSON.", "Validate each amount and output the approved breakdown."). Do NOT implement verification or parsing in all_code (no json.loads, re, or try/except to fix LLM output). Use the LLM to do verification and output clean results.

LLM_Generator nodes: Provide full System Prompt and User Prompt in prompt_template; put any validation/format rules there. all_code must only: extract inputs, build node_messages from prompt_template, call self.llm_client.chat, return {{output_key: response}}. No code that parses or validates the response.

Retrieval_RAG nodes: logic_description must state what to retrieve and how to summarize. prompt_template must include System Prompt and User Prompt; use a placeholder like {{retrieved_context}} or {{retrieved_chunks}} for the search result. all_code must only: build query from inputs, call self.search_engine.multi_turn_search(query), build node_messages from prompt_template with retrieved content, call self.llm_client.chat, return output. No code that parses or validates the response.

Retrieval_RAG: Design so you do NOT retrieve the question itself; retrieve only related knowledge (e.g. laws, case law, background) needed to answer. State this in logic_description and prompt_template.

4. Design principles:

Use only LLM_Generator and Retrieval_RAG.

Completeness over brevity: Ensure the pipeline has enough nodes for the task. List all logical steps the task requires (from task description and strategy analysis), then create one node (or more) for each step. When in doubt, add a dedicated node rather than overloading one node with multiple responsibilities. Too few nodes often lead to incomplete or poor results.

Each node has a single responsibility. Dependencies form a DAG.

Use LLM_Generator for reasoning, generation, extraction, validation, and any step that would otherwise need "code" (e.g. ask LLM to output structured JSON or numbers).

Use Retrieval_RAG when external knowledge retrieval (search) is needed, then LLM to summarize or use the retrieved context.

5. Output format:

Provide a JSON object with this structure:

{{

"pipeline_description": "Overall description of the pipeline and how nodes work together",

"nodes": [{{

"node_name": "...",

"node_type": "LLM_Generator or Retrieval_RAG only",

"description": "...",

"dependencies": ["..."],

"input": ["..."],

"output": ["..."],

"constraints": "...",

"implementation": {{ "logic_description": "...", "prompt_template": "...", "tools_needed": ["Search"] for Retrieval_RAG, [] for LLM_Generator }},

"all_code": "Minimal code only: input extraction, then LLM call(s) or search+LLM, then return output_data. No verification/parsing blocks."

}}, ...],

"Connections": "Complete Python code for def execute_pipeline(self, initial_input_data): ... Execute nodes in dependency order; collect inputs from initial_input_data or results; call self.NodeName(input_data); store outputs; return final output. Import json if needed."

}}

Remember:

Carefully check that the task needs are fully covered by nodes: Before outputting, verify you have a node for every required step (e.g. input understanding, retrieval if needed, reasoning, synthesis, final answer). The number of nodes should be sufficient to solve the task completely—do not output a pipeline with too few nodes.

Use only LLM_Generator and Retrieval_RAG.

All verification and parsing must be done by the LLM: write rules and output-format requirements in the prompt_template; do not write code to verify or parse LLM output (no json.loads, re, try/except for validation/parsing in all_code).

all_code must be minimal: read input -> (LLM call or search+LLM) -> return output. No code that checks or parses the LLM response. Dependencies must form a valid DAG. Use task samples to align input/output formats.

For "Connections": generate the pipeline execution function that runs nodes in dependency order and passes data correctly.

Figure 17: Prompt for Node Generation Part 2.

Prompt for Node Optimization

System_prompt:
You are an expert system optimizer and code reviewer. Your task is to analyze a node in a multi-agent pipeline that has the lowest reward and optimize its internal structure to improve performance. All optimizations must be achieved via the LLM. You may: (1) improve existing LLM prompts, (2) introduce new LLM calls where needed, (3) optimize how multiple LLM calls within the same node communicate and interact—e.g. what is passed between calls, in what format, in what order, and how results are aggregated. Do NOT add Python code for rules, regex, normalization, or filtering—fix shortcomings by prompt engineering or by adding/adjusting LLM calls and their communication, not by code.
User_prompt: Question: <question>
Expected Answer: <answer>
 # Node to Optimize
Node Name: node_name
Node Type: node_type
Node Description: node_description
Node Reward: node_reward (This is the lowest reward, indicating poor performance)
Node Position: Node node_index + 1 in the pipeline
 # Current Node Implementation
Implementation Details: {json.dumps(node_implementation, ensure_ascii=False, indent=2)}
Current Code: {node_all_code}
 # Pipeline Context
All Intermediate Outputs (to understand the data flow; when multiple samples exist, each [Sample N] block's node outputs correspond to the [Sample N] Question/Answer in Task Context above): {intermediate_context}
 # Analysis Task
Based on the question, expected answer, and the intermediate outputs from all nodes, analyze why this node has the lowest reward and provide optimization suggestions. Analysis Steps:
1. Identify the Problem: What is the node's current output? (from intermediate_outputs) How does it differ from what's expected? What specific issues are causing the low reward?
2. Root Cause Analysis: Is the prompt (for LLM_Generator/Retrieval_RAG) clear and specific enough? Are the LLM calls structured optimally? If the node has multiple LLM calls, is the communication between them effective—e.g. is the handoff from one call to the next clear, in a good format, and in the right order? Are there missing or redundant steps? Is the implementation handling all cases correctly? Is the retrieval (for Retrieval_RAG) getting relevant information? Are there any logical errors or missing validations?
3. Optimization Strategy: {optimization_focus}
CRITICAL: Fix shortcomings via LLM, not code: You may (1) improve existing prompts, (2) introduce new LLM calls (e.g. a refinement or validation step), (3) optimize inter-LLM communication when a node has multiple LLM calls—e.g. clarify what each call receives from the previous one, improve the handoff format in the prompt, reorder or add calls so the flow is clearer. Do NOT add Python code for rule-based checks, regex, normalization, or filtering. The code should remain minimal: prepare inputs → call LLM(s), passing outputs between calls as needed → return output.
 # Output Format
Provide a JSON object with the following structure:
{{
"analysis": {{ "problem_identification": "Detailed description of what's wrong with the current node", "root_cause": "Analysis of why the node is performing poorly", "optimization_strategy": "Specific strategy to improve the node" }},
"optimized_implementation": {{ "prompt_template": "Updated prompt template (marked as System Prompt: and User Prompt:). Keep original if no changes needed.", "tools_needed": "Updated tools_needed (for Retrieval_RAG nodes). Keep original if no changes needed.", "logic_description": "Updated logic description explaining the optimization" }},
"optimized_all_code": "Complete updated code for the node following the code_template structure. MUST be complete and runnable. Output the code in the same format as the original code.
"optimization_explanation": "Detailed explanation of what was optimized and why".
}}
IMPORTANT:
Fix any identified problems by improving the prompt, adding or reordering LLM calls, or improving how multiple LLM calls in the same node communicate (what is passed, in what format). Do NOT add Python code for validation, regex, normalization, rule-based filtering, or parsing of LLM output. optimized_all_code must stay minimal: get inputs → call LLM(s), passing outputs between calls as needed → return output.
For LLM_Generator: Improve prompt_template and/or the number and sequence of LLM calls; if there are multiple calls, ensure each call's prompt clearly receives and uses the outputs of previous calls. Do not add code to parse or validate responses.
For Retrieval_RAG: Improve the summarization prompt and query construction; do not add code to filter or normalize retrieved content—instruct the LLM to do it in the prompt.
The optimized_all_code MUST be complete and runnable but MUST NOT contain extra validation/parsing/regex/filtering code.
If you add, remove, or reorder LLM calls, or change how they communicate (handoff format/order), explain the reasoning in logic_description.

Figure 18: Prompt for Node Optimization.