AutoSwarm: Training LLM to Self-orchestrate via Reinforcement Learning

Anonymous ACL submission

Abstract

Recent advances in large language models (LLMs) have enabled new agentic workflows where multiple LLMs collaborate in specialized roles. Current approaches to designing these workflows face key limitations: manual design requires substantial human expertise, while existing automated frameworks struggle with optimization efficiency and task adaptability. To address these challenges, we present AutoSwarm, a novel system that trains an LLM orchestrator through reinforcement learning to generate executable code. The generated code can be directly executed in a workflow runtime environment, with the orchestrator learning end-to-end through a reward mechanism that optimizes both performance and efficiency. AutoSwarm outperforms existing automated workflow methods, achieving a 1.91% accuracy improvement on reasoning benchmarks. The system also shows robust generalization, with a 1.25% performance gain on out-of-distribution tasks. Our work explores a promising direction for learning-based workflow orchestration.[1]

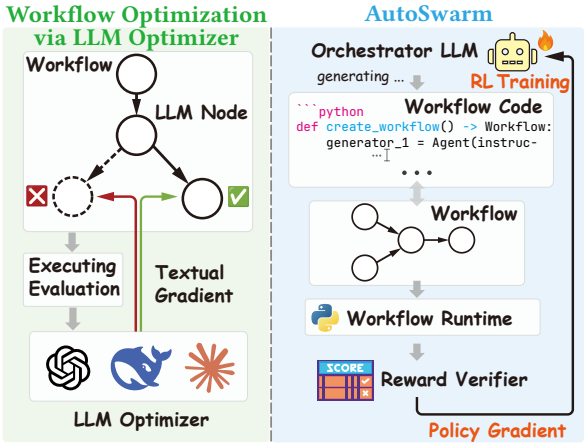

Figure 1: Comparison between existing automated workflow optimization via LLM optimizer (left) and AutoSwarm (right). Existing approaches rely on iterative refinement through LLM-based optimization, which can be inefficient and limited in generalization. In contrast, AutoSwarm employs reinforcement learning to train an LLM orchestrator that generates complete workflows in a single forward pass, enabling continuous improvement and better adaptation to novel scenarios.

1 Introduction

The emergence of large language models (LLMs) has revealed a potential pathway toward artificial general intelligence (AGI), driven primarily by the success of scaling laws (Brown et al., 2020; Hoffmann et al., 2022; Chowdhery et al., 2022). By systematically expanding model parameters, training data volume, and computing resource, LLMs have exhibited remarkable reasoning capabilities that not only approach but in some cases exceed human-level performance (OpenAI, 2024a; DeepSeek-AI, 2024; Phan et al., 2025). However, as high-quality pretraining data sources become increasingly exhausted (Abdin et al., 2024; Alemohammad et al., 2023), the traditional pre-training scaling paradigm faces fundamental limitations. The data scarcity has prompted researchers to actively explore alternative paradigms to amplify LLM intelligence. Recent advances span multiple directions: large-scale reinforcement learning (OpenAI, 2024b; DeepSeek-AI, 2025; Kimi-Team, 2025), test-time enhancement through search algorithms like Monte Carlo Tree Search (MCTS) (Zhao et al., 2023; Zhang et al., 2024a; Jiang et al., 2024), and agentic workflows that leverage collective intelligence through role-based collaboration (Li et al., 2023; Wu et al., 2023).

Among these approaches, agentic workflows represent a promising direction for enhancing LLM capabilities. The orchestration of multiple agents in complementary roles enables emergent cognitive capabilities through their structured interactions and collaborative problem-solving, analogous to the collective intelligence observed in human organizations. Current approaches to designing

[1]Codes and datasets are available at https://anonymous.4open.science/r/Agentic-7EEA.

such workflows can be categorized into two main paradigms: (1) manual design by human experts, which offers precise control but suffers from limited scalability and requires substantial human expertise and effort; and (2) automated optimization frameworks such as GPTSwarm (Zhuge et al., 2024), ADAS (Hu et al., 2024), EvoMac (Hu et al., 2025) and AFLOW (Zhang et al., 2025). These automated approaches share a fundamental design principle: they employ LLMs as optimization engines to iteratively improve workflows by using feedback from execution outcomes and making adjustments accordingly. (See Figure 1 left)

However, these automated workflow design approaches suffer from several limitations. First, the effectiveness of LLMs in optimizing workflows solely through textual gradient is questionable, particularly considering that LLMs likely lack training data in workflow optimization scenarios. Second, using LLMs as optimizers requires multiple steps of discrete optimization within the search space, resulting in suboptimal efficiency. Most critically, these systems are fundamentally limited to searching for workflows within predefined problem subsets, and cannot generalize beyond their initial design. Unlike learning-based approaches, they lack the capability to leverage training data to improve their workflow optimization abilities over time, making them unable to learn and adapt to new scenarios or accumulate experience across different workflow optimization tasks.

To overcome existing limitations, we introduce AutoSwarm, a system that employs an LLM as an orchestrator to adaptively generate workflows based on user inputs. Specifically, we implement a Python-based workflow framework where the orchestrator LLM generates executable code defining the roles, dependencies, and execution logic of multiple specialized LLMs. This generated code can be directly executed in our implemented sandbox environment to obtain workflow outputs. We design a comprehensive reward function based on output correctness and workflow efficiency, and utilize reinforcement learning to optimize the orchestrator's workflow generation capabilities.

Our proposed AutoSwarm offers several key advantages: (1) it enables end-to-end learning of workflow generation strategies through direct optimization of execution outcomes, (2) it maintains efficiency by generating complete workflows in a single forward pass rather than through iterative refinement, and (3) it accumulates experience across tasks to continuously improve its workflow design capabilities, allowing for better generalization to novel scenarios. By combining the flexibility of LLM-based generation with the systematic optimization capabilities of reinforcement learning, AutoSwarm represents a significant step forward in automated workflow design.

Our extensive experiments demonstrate the effectiveness of AutoSwarm across multiple dimensions. When evaluated on five mathematical benchmarks, AutoSwarm achieves an average accuracy of 47.36%, outperforming current state-of-the-art automated workflow design approaches by 1.91%. The system also shows strong generalization capabilities, successfully transferring its learned orchestration strategies to various out-of-distribution domains, with consistent improvements averaging 1.25% over baseline models. Through detailed analysis of training dynamics and case studies, we observe the orchestrator's evolution from basic linear workflows to sophisticated, problem-specific architectures, demonstrating its ability to learn and adapt complex workflow optimization strategies through reinforcement learning.

2 Methods

2.1 Preliminaries

Workflow Definition We formally define a workflow W as a directed acyclic graph $\mathcal{A} = (\mathcal{M} \cup \perp, \mathcal{E})$, where language models serve as nodes and $\perp$ marks the workflow end. The set $\mathcal{M} = \{M_1, \ldots, M_n\}$ represents specialized language models that write outputs to a shared memory $\mathcal{S}$. The edge set $\mathcal{E}$ represents model dependencies, where $(M_i, M_j) \in \mathcal{E}$ indicates M_j executes after M_i completes. For any $M_i \in \mathcal{M}$ without outgoing edges, we add $(M_i, \perp) \in \mathcal{E}$. The workflow terminates at $\perp$ and returns its predecessor's output.

Each model $M_i \in \mathcal{M}$ is defined by (p_i, τ_i), where prompt p_i defines the LLM's role-specific instructions and temperature τ_i controls sampling randomness.

Inter-Model Communication The workflow implements a centralized shared memory (scratchpad) $\mathcal{S}$ that maintains contextual information across models. Unlike prior work (Zhang et al., 2025) that only passes final results between models, the complete reasoning process of each model is preserved in $\mathcal{S}$, enabling subsequent models to build upon their predecessors' reasoning chains. While questions about the scalability of this shared mem-

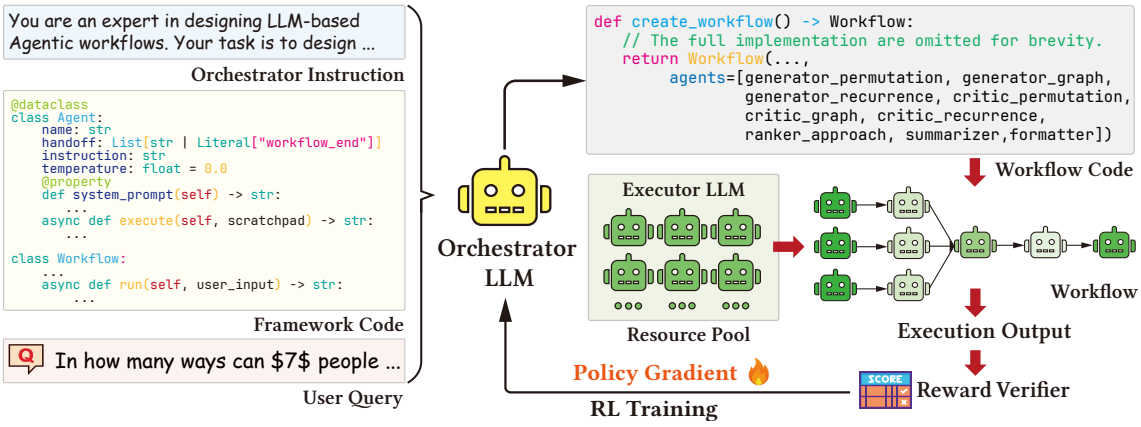

Figure 2: The architecture of AutoSwarm enables dynamic workflow synthesis through reinforcement learning. An orchestrator LLM generates executable workflow code, which is executed by specialized executor LLMs. The system improves through reinforcement learning, with the orchestrator receiving rewards based on solution correctness and workflow efficiency.

ory architecture exist, they are beyond this paper's scope.

Overview AutoSwarm introduces a novel approach to LLM-based workflow synthesis by training an LLM orchestrator through reinforcement learning. As illustrated in Figure 2, the system consists of three key components: (1) a code-based workflow framework for generating executable Python code, (2) a resource pool of executor LLMs for computation, which are instantiated as workflow nodes based on the orchestrator-generated code, and (3) a reinforcement learning framework that optimizes the orchestrator. Given an input problem, the orchestrator generates workflow code to coordinate multiple specialized LLMs, which is then executed across the resource pool. Through rewards based on solution correctness and efficiency, the orchestrator learns to generate increasingly effective workflows over the training process.

2.2 LLM-Driven Workflow Synthesis

AutoSwarm introduces a novel meta-programming paradigm where a specialized orchestrator LLM $\mathcal{O}$ dynamically generates executable code that expresses problem-specific workflows. Given an input problem $x \in \mathcal{X}$ from the problem space, AutoSwarm leverages orchestrator LLM $\mathcal{O} : \mathcal{X} \to \mathcal{W}$, where $\mathcal{W}$ represents the executable Python code of the workflow W.

Workflow Code Generation The orchestrator LLM $\mathcal{O}$ generates Python code that implements workflows using pre-defined code framework $\mathcal{F}$, which provides two core classes: `Agent` and `Workflow`. The `Agent` class encapsulates each language model's characteristics, including its prompt p_i, temperature τ_i, and dependency relationships e_i with other models. The `Workflow` class orchestrates the execution logic outlined in Section 2.1, managing the directed flow of information between models.

Specifically, given a input problem, the orchestrator LLM acts as a meta-programmer, analyzing the problem and synthesizing appropriate code. The orchestrator LLM combines the following components into its context:

- **Framework Code** $\mathcal{F}$ - The core classes and utilities that serve as building blocks for the generated workflow code
- **Orchestrator Instruction** $I_{\mathcal{O}}$ - Specific guidance for the orchestrator on how to construct the workflow code
- **User Query** x - The specific problem that needs to be solved using the workflow

The orchestrator LLM then generates the executable workflow code by implementing a `create_workflow` function that will return a `Workflow` object, which can be formulated as:

$$\mathcal{W} = \mathcal{O}(\text{Context}(\mathcal{F}, I_{\mathcal{O}}, x)) \quad (1)$$

Workflow Runtime After the Orchestrator LLM generates the workflow code $\mathcal{W}$ for a specific problem, the code is delegated to a resource pool containing multiple executor LLMs for actual execution. The architecture of AutoSwarm mirrors established distributed computing frameworks like

Algorithm 1 Inference Procedure of AUTOSWARM

Require:
User query $x \in \mathcal{X}$
Framework code $\mathcal{F}$
Orchestrator LLM $\mathcal{O}$ and its instruction $I_{\mathcal{O}}$
Pool of executor LLMs $\mathcal{P}$
Ensure: Final workflow output
Phase I: Workflow Code Generation
function GENWORKFLOW($x, \mathcal{F}, I_{\mathcal{O}}, \mathcal{O}$)
  `context` $\xleftarrow{\text{compose}}$ Context($\mathcal{F}, I_{\mathcal{O}}, x$)
  $\mathcal{W} \leftarrow \mathcal{O}$(`context`)
  return $\mathcal{W}$ ▷ Workflow Code
end function
Phase II: Workflow Execution
function EXECUTEWORKFLOW($\mathcal{W}, \mathcal{P}$)
  $\mathcal{S} \leftarrow \emptyset$ ▷ Shared scratchpad
  $W \xleftarrow{\text{instantiation}} \mathcal{W}$
  $\mathcal{M} \leftarrow$ InitializeModels(W)
  `ready` $\leftarrow$ GetWorkflowStartModel(W)
  while `ready` $\neq \emptyset$ **do**
    for all $M_i \in$ `ready` **in parallel do**
      `executor` $\leftarrow$ GetExecutor($\mathcal{P}$)
      `input` $\leftarrow$ PrepareInput($M_i, \mathcal{S}$)
      `output` $\leftarrow$ `executor`.Run(`input`)
      if IsTerminalNode(W, M_i) **then**
        return `output`
      end if
      UpdateMemory($\mathcal{S}$, M_i, `output`)
    end for
    `ready` $\leftarrow$ GetNextModels($W, \mathcal{M}$)
  end while
end function

Spark, where the Orchestrator LLM assumes a management role analogous to a master node - handling high-level tasks like workflow decomposition and orchestration - while the computational workload is efficiently distributed across the executor LLMs that act as worker nodes. This architecture achieves enhanced reasoning capabilities through its synergistic combination of high-level workflow planning and parallel execution, while maintaining efficient resource utilization.

AutoSwarm Inference Procedure Algorithm 1 formalizes the complete inference procedure of AutoSwarm, which consists of two main phases outlined above. During the workflow code generation phase, the orchestrator LLM analyzes the input problem and generates executable workflow code using the provided framework. Subsequently, in the workflow execution phase, the generated workflow is instantiated and executed across the executor LLM pool, with results aggregated in the shared memory space until the terminal node is reached.

2.3 Orchestrator Reinforcement Learning

In this section, we present our reinforcement learning framework that enables systematic training of workflow orchestrators. The framework consists of two key components: (1) a reward mechanism balancing correctness and efficiency (Section 2.3.1), and (2) an end-to-end training pipeline using the group-based policy optimization (GRPO) algorithm (Shao et al., 2024) (Section 2.3.2). Below we detail these components.

2.3.1 Reward Modeling for Orchestrator

The reward signals used for training orchestrator LLM $\mathcal{O}$ are derived from both workflow correctness and solution efficiency.

Workflow Correctness Reward A workflow's correctness is evaluated through a two-stage verification process: (1) workflow validity check and (2) execution result verification. The correctness reward $r_{\text{cor}}(x, \mathcal{W})$ for workflow $\mathcal{W}$ is defined as:

$$r_{\text{cor}}(x, \mathcal{W}) = \begin{cases} 1 & \text{if valid}(\mathcal{W}) \wedge \mathcal{W}(x) = y^* \\ -0.5 & \text{if valid}(\mathcal{W}) \wedge \mathcal{W}(x) \neq y^* \\ -1 & \text{if } \neg\text{valid}(\mathcal{W}) \end{cases} \tag{2}$$

The workflow validity (valid($\mathcal{W}$)) is checked using a Python interpreter with predefined rules, including syntactic correctness, architectural patterns compliance, and runtime limit constraints. For valid workflows, the execution results are then compared against ground truth answers (y^*) through rule-based verification.

Workflow Efficiency Reward The efficiency reward $r_{\text{effcy}}(x, \mathcal{W})$ is formulated as:

$$r_{\text{effcy}} = \begin{cases} -\gamma & \text{if } |\mathcal{M}| > M_{\max} \\ \alpha \sin^2\left(\frac{\pi |\mathcal{M}|}{2M_{\max}}\right) & \text{if } |\mathcal{M}| \leq M_{\max} \wedge r_{\text{cor}}(x, \mathcal{W}) = 1 \\ 0 & \text{if } |\mathcal{M}| \leq M_{\max} \wedge r_{\text{cor}}(x, \mathcal{W}) \neq 1 \end{cases} \tag{3}$$

where $|\mathcal{M}|$ is the number of models, $M_{\max}$ is the maximum allowed model count, and α and γ are hyperparameters. The sinusoidal decay function maintains stable gradients while optimizing for efficiency as a secondary objective to correctness.

Composite Reward Function The final reward combines correctness and efficiency:

$$r = r_{\text{cor}}(x, \mathcal{W}) + \lambda r_{\text{effcy}}(x, \mathcal{W}) \tag{4}$$

where λ controls the trade-off between correctness and efficiency.

2.3.2 Optimizing the Orchestrator Policy

With our designed reward defined in Equation (4) that jointly optimizes solution correctness and resource efficiency, we optimize the orchestrator using GRPO as our training algorithm, which offers an alternative to Proximal Policy Optimization (PPO) (Schulman et al., 2017). While proven effective by many previous research efforts (DeepSeek-AI, 2025; Kimi-Team, 2025), GRPO eliminates the need for a critic to estimate the value function. Instead, it uses the average return of multiple sampled outputs, produced in response to the same question, as the baseline.

Specifically, for each input user problem x, GRPO samples a group of workflow code $\{\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_G\}$ from the old policy θ_{old}, which represents the parameters of orchestrator LLM $\mathcal{O}$ in the previous policy iteration. Each workflow code represents a different approach to decomposing and solving the given problem, potentially varying in their model composition, dependency structure, and resource utilization patterns. The orchestrator then optimizes its policy by maximizing the following objective:

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}_{x\sim\mathcal{D},\{\mathcal{W}_i\}_{i=1}^{G}\sim\theta_{\text{old}}} \frac{1}{G}\sum_{i=1}^{G}\Big[\min\Big(\frac{\pi_\theta(\mathcal{W}_i|x)}{\pi_{\theta_{\text{old}}}(\mathcal{W}_i|x)}\hat{A}_i(x), \text{clip}\left(\frac{\pi_\theta(\mathcal{W}_i|x)}{\pi_{\theta_{\text{old}}}(\mathcal{W}_i|x)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_i(x)\Big) - \beta\mathbb{D}_{\text{KL}}\Big] \tag{5}$$

$$\mathbb{D}_{\text{KL}} = \frac{\pi_{\text{ref}}(o_i|q)}{\pi_\theta(o_i|q)} - \log\frac{\pi_{\text{ref}}(o_i|q)}{\pi_\theta(o_i|q)} - 1 \tag{6}$$

where $\mathcal{D}$ is the distribution of RL training data, ϵ and β are hyper-parameters, and $\hat{A}_i$ denotes the group-normalized advantage, computed using a group of rewards corresponding to the output workflow codes within each group:

$$\hat{A}_i = \frac{r_i - \text{mean}(\{r_1, r_2, \ldots, r_G\})}{\text{std}(\{r_1, r_2, \ldots, r_G\})} \tag{7}$$

3 Experiments

This section investigates three research questions:

- **RQ1:** How does AutoSwarm compare to existing workflow orchestration methods? (Section 3.1)
- **RQ2:** How effective is our Orchestrator versus state-of-the-art LLMs, and how well does it generalize? (Section 3.2)
- **RQ3:** How does the Orchestrator evolve during training? (Section 3.3)

Implementation Details We choose Qwen2.5-Coder-32B-Instruct as the base model for the reinforcement learning of the orchestrator. The GRPO algorithm is implemented upon the OpenRLHF framework. We sample 8 responses for each query in GRPO. Each episode comprising 1024 samples. We utilize a training batch size of 128 and optimize using a learning rate of 5e-7. The reinforcement learning process exhibits convergence at approximately 90 steps, with reward signals reaching saturation. The checkpoint at step 96 was selected for subsequent evaluation experiments. Comprehensive experimental parameters and hyperparameter settings are detailed in Appendix A.

Workflow Execution Environment We utilize `Qwen2.5-7B-Instruct` as the executor LLM. During the RL training process, we form an executor resource pool consisting of 16 executor LLM instances. While obtaining workflow execution feedback accounts for approximately 15% of the time in a complete RL training step, this overhead can be reduced through increased parallelization or overlapping with other RL steps to achieve improved efficiency.

RL Data Construction We sampled 3,000 problems from the MATH500 dataset's training set for RL training, following these sampling criteria: The Executor LLM must achieve at least one correct answer in 16 repeated sampling attempts with a temperature of 0.6, while maintaining an accuracy rate below 0.9. This ensures that the problems are solvable by the Executor LLM while still presenting sufficient challenge.

3.1 Comparison on Workflow Optimization

Baselines We evaluate AutoSwarm by comparing it with both traditional and state-of-the-art approaches. For traditional methods, we include Chain-of-Thought prompting (CoT) (Wei et al., 2023), Self-Consistency (Wang et al., 2023), and more advanced techniques like MultiPersona Debate (Wang et al., 2024b), Self-Refine (Madaan et al., 2023) and MedPrompt (Nori et al., 2023). Additionally, we benchmark against recent automated workflow optimization frameworks including ADAS (Hu et al., 2024) and AFLOW (Zhang et al., 2024b) to provide a comprehensive comparison across the spectrum of existing solutions.

Table 1: Comparison of AutoSwarm with existing workflow design methods. Results show accuracy (%) on mathematical benchmarks.

Method	**Benchmarks**					**Avg.**
	MATH500	AIME 24	OlympiadBench	AMC 23	Minerva Math	
CoT (Wei et al., 2023)	74.40	13.33	37.19	57.50	30.51	42.59
CoT SC (Wang et al., 2023)	75.60	16.67	38.37	60.00	31.62	44.45
MedPrompt (Nori et al., 2023)	74.80	13.33	37.78	60.00	31.25	43.43
MultiPersona (Wang et al., 2024b)	75.80	16.67	38.67	62.50	31.99	45.12
Self Refine (Madaan et al., 2023)	72.80	13.33	36.44	55.00	29.78	41.47
ADAS (Hu et al., 2024)	71.80	6.67	35.85	55.00	29.41	39.75
AFLOW (Zhang et al., 2024b)	76.60	16.67	39.11	62.50	32.35	45.45
AutoSwarm (Ours)	**78.20**	**20.00**	**40.15**	**65.00**	**33.46**	**47.36**

Table 2: Comparison of AutoSwarm with state-of-the-art LLMs as workflow orchestrators. All methods use Qwen2.5-7B-Instruct as the executor model.

Orchestrator + Qwen2.5-7B-Instruct as Executor	**Benchmarks**					**Avg.**
	MATH500	AIME 24	OlympiadBench	AMC 23	Minerva Math	
Qwen2.5-Coder-32B-Instruct (Hui et al., 2024)	74.20	10.00	36.89	55.00	31.25	41.47
DeepSeek V3 (DeepSeek-AI, 2024)	74.60	13.33	37.33	57.50	31.62	42.88
GPT-4o (OpenAI, 2024a)	74.80	10.00	37.48	57.50	30.88	42.13
Claude-3.5-Sonnet (Anthropic, 2024)	75.60	13.33	38.07	60.00	31.99	43.80
AutoSwarm (Ours)	**78.20**	**20.00**	**40.15**	**65.00**	**33.46**	**47.36**

Implementation Details To ensure fair comparison, we maintain identical model configurations between our method and baselines. Specifically, for both ADAS and AFLOW, we use Claude-3.5-Sonnet as their optimizer LLM, matching their original configurations, and Qwen2.5-7B-Instruct as their executor LLM, identical to our execution model for fair comparison. All other hyperparameters, including the maximum number of iterations, strictly follow the specifications in the AFLOW paper.

Dataset For comprehensive evaluation, we utilize five mathematical benchmarks: AMC 23, AIME 24, MATH500 (Hendrycks et al., 2021), OlympiadBench (He et al., 2024) (using text-only math problems) and Minerva Math (Lewkowycz et al., 2022). These benchmarks span different mathematical domains and difficulty levels to ensure comprehensive evaluation.

Results Table 1 presents a comparison of AutoSwarm against existing workflow design methods across mathematical benchmarks. Our approach achieves an average accuracy of 47.36%, consistently outperforming both manual workflow design methods and automated workflow optimization frameworks. AutoSwarm demonstrates significant advantages over existing workflow design approaches. First, compared to manual workflow design methods, AutoSwarm shows a substantial improvement of 4.77%, highlighting the limitations of human-designed workflow patterns. Second, against automated workflow optimization frameworks, AutoSwarm outperforms the strongest baseline AFLOW by 1.91 percentage points on average, demonstrating the advantages of end-to-end reinforcement learning approaches over methods that use LLMs as optimizers. This improvement suggests that direct policy optimization through RL enables more effective workflow orchestration compared to discrete optimization steps performed by LLM-based optimizers.

3.2 Orchestrator Performance Comparison

Settings A critical research question we investigate is the comparative performance between AutoSwarm and existing state-of-the-art language models in orchestration tasks. Specifically, we examine whether contemporary LLMs can outperform AutoSwarm as Orchestrators, and quantify the improvements achieved through our reinforcement learning approach compared to the base model.

For comparative evaluation of workflow orchestration capabilities, we utilized a set of contem-

porary LLMs as orchestrator baselines, including Qwen2.5-Coder-32B-Instruct (Hui et al., 2024), Deepseek-V3 (DeepSeek-AI, 2024), GPT-4o (OpenAI, 2024a) and Claude-3.5-Sonnet (Anthropic, 2024), benchmarked against AutoSwarm's orchestrator. All evaluations were conducted using Qwen2.5-7B-Instruct as the Executor model.

Comparison Analysis The experimental results reveal key insights about orchestration capabilities of current language models. As shown in Table 2, while state-of-the-art LLMs demonstrate basic orchestration abilities, they consistently underperform compared to our RL-trained orchestrator across all benchmarks. This gap stems primarily from insufficient orchestration-specific training in existing LLMs - despite their strong general capabilities, these models lack exposure to complex workflow orchestration scenarios. Even advanced models like DeepSeek-V3 (42.88% average accuracy) and Claude-3.5-Sonnet (43.80% average accuracy) show limitations in this specialized domain.AutoSwarm addresses these limitations through targeted reinforcement learning. The results demonstrate that AutoSwarm's framework (47.36% average accuracy) significantly enhances orchestration capabilities, achieving a 5.8% improvement over its base LLM Qwen2.5-Coder-32B-Instrcut. These consistent improvements across all benchmarks highlight the effectiveness of our reinforcement learning paradigm in elevating the model's workflow orchestration intelligence.

Generalization Analysis To investigate generalization beyond mathematical problems, we evaluated our RL-trained orchestrator on three domains (Physics, Chemistry, and Law) from MMLU-Pro (Wang et al., 2024a). The results in Table 3 show that AutoSwarm generalizes well to these out-of-distribution tasks, achieving improvements in Physics (+1.77%), Chemistry (+0.71%), and Law (+1.27%). This consistent improvement (1.25% on average) suggests that the learned orchestration strategies represent general workflow optimization principles that transfer across domains.

3.3 Orchestrator Training Dynamics Analysis

RL Training Dynamics Figure 3 illustrates the training dynamics of AutoSwarm's reinforcement learning process. The reward curve shows consistent improvement in the Orchestrator's performance, characterized by two distinct phases: (1) an initial rapid improvement phase (steps 1-32) with sharp reward increases, demonstrating quick adaptation to basic orchestration strategies; and (2) a steady optimization phase (steps 32-96) with gradual improvements as the model refines its techniques. Periodic evaluations on the MATH500 test set validate that reward improvements correlate with actual performance gains, confirming the effectiveness of our reward design.

Table 3: Accuracy (%) on Out-of-distribution evaluation benchmarks. Physics, Chemistry and Law are from MMLU-Pro dataset. ↑ indicates the improvement of AutoSwarm after RL training.

Method	Benchmarks			Avg.
	Physics	Chemistry	Law	
Executor	59.28	55.65	31.79	48.91
w/o RL	59.43	55.74	31.61	48.93
AutoSwarm	61.20 ↑1.77	56.45 ↑0.71	32.88 ↑1.27	50.18 ↑1.25

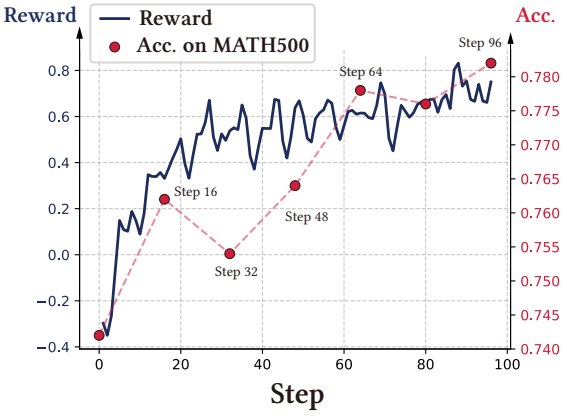

Figure 3: RL Training curve of AutoSwarm, with training steps on the x-axis and reward on the y-axis. Checkpoints are saved every 16 steps to compute accuracy on the MATH500 test set.

Case Study To illustrate the orchestrator's learning progression, we analyze workflow evolution during training. As shown in Figure 4, examining a test instance from MATH500, the orchestrator demonstrates clear development from basic to sophisticated workflows. At step 16, it generates simple linear workflows with straightforward solutions. By step 96, it produces complex, problem-specific architectures with parallel generator components and generator-critic frameworks. This evolution enables the orchestrator to adapt workflow structures based on problem complexity and requirements, consistently demonstrating its advancement in optimizing problem-specific workflows.

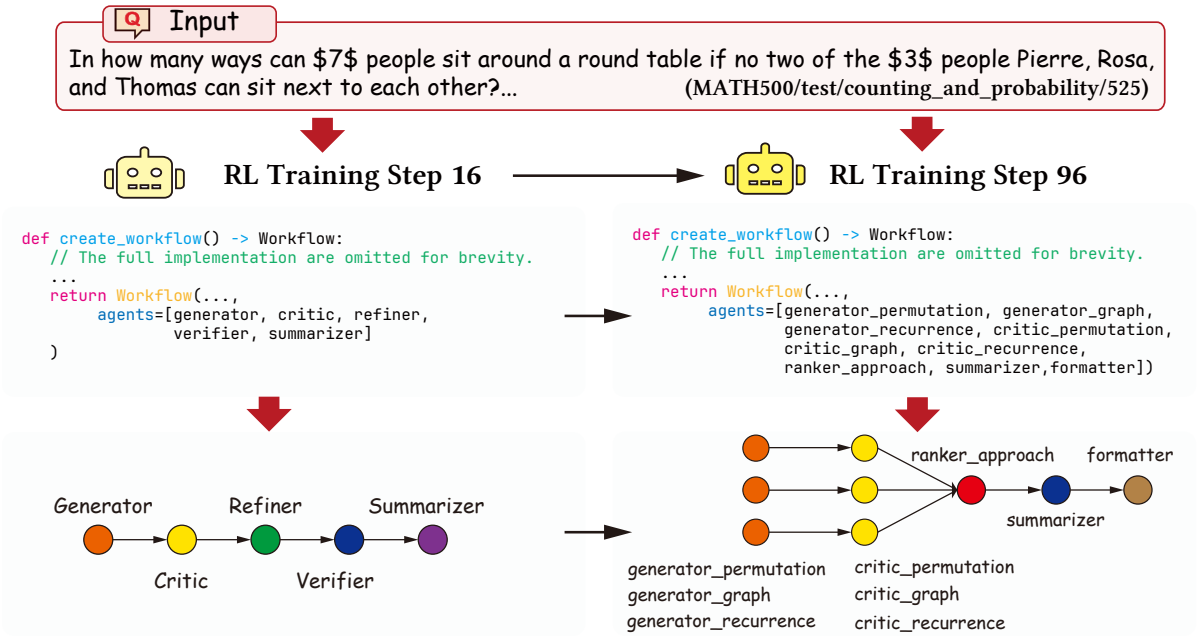

Figure 4: Case Study: Evolution of workflow sophistication in workflows generated by the Orchestrator LLM, demonstrated on a test case (`counting_and_probability-525`) from MATH500 test set. The transformation from elementary linear patterns (Step 16) to sophisticated architectures including parallel generator and generator-critic frameworks (Step 96) illustrates the model's architectural learning trajectory.

4 Related Works

Agentic Workflow Orchestration Recent research has highlighted the emergence of agentic workflows - predefined processes leveraging multiple LLM invocations to accomplish complex tasks through domain expertise. These workflows span from universal problem-solving methods (Wang et al., 2023; Madaan et al., 2023; Wang et al., 2024b) to specialized solutions like data analysis (Xie et al., 2024) and software development (Hong et al., 2024). The automation and optimization of these workflows has emerged as a critical research direction. Key examples include GPTSwarm (Zhuge et al., 2024), which represents LLM-based workflows as computational graphs with optimizable node-level prompts and edge-level orchestration, ADAS (Hu et al., 2024), which pioneers code-based structures but faces efficiency challenges due to simplified experience representations, and AFLOW (Zhang et al., 2024b), which introduces named nodes and MCTS-based optimization but is limited by its discrete optimization approach.

AutoSwarm advances this field through two key innovations: end-to-end reinforcement learning for workflow optimization based on execution outcomes, and an orchestrator that generalizes across tasks through accumulated experience. These enable more effective workflows while maintaining single-pass generation efficiency.

5 Conclusion

This paper introduces AutoSwarm, a novel approach for workflow orchestration through reinforcement learning of LLM orchestrators. Our key contributions include an end-to-end framework for generating executable workflow code and a reward mechanism optimizing both correctness and efficiency. Experimental results show AutoSwarm achieves 47.36% accuracy on mathematical benchmarks, outperforming state-of-the-art by 1.91%, while demonstrating strong generalization with 1.25% improvements on out-of-distribution tasks.

6 Limitations

While AutoSwarm demonstrates strong performance, some limitations remain. The current implementation requires substantial computational resources for reinforcement learning. Additionally, the system's performance may vary across different domains and problem types. Future work could explore more efficient training methods and expanded application scenarios.

References

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. 2024. Phi-4 technical report.

Sina Alemohammad, Josue Casco-Rodriguez, Lorenzo Luzi, Ahmed Imtiaz Humayun, Hossein Babaei, Daniel LeJeune, Ali Siahkoohi, and Richard G. Baraniuk. 2023. Self-consuming generative models go mad.

Anthropic. 2024. Introducing claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways.

DeepSeek-AI. 2024. Deepseek-v3 technical report.

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

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. 2024. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems.

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

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. Metagpt: Meta programming for a multi-agent collaborative framework.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2025. Self-evolving multi-agent networks for software development. In *The Thirteenth International Conference on Learning Representations*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report.

Jinhao Jiang, Zhipeng Chen, Yingqian Min, Jie Chen, Xiaoxue Cheng, Jiapeng Wang, Yiru Tang, Haoxiang Sun, Jia Deng, Wayne Xin Zhao, Zheng Liu, Dong Yan, Jian Xie, Zhongyuan Wang, and Ji-Rong Wen. 2024. Enhancing llm reasoning with reward-guided tree search.

Kimi-Team. 2025. Kimi k1.5: Scaling reinforcement learning with llms.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving quantitative reasoning problems with language models.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback.

Harsha Nori, Yin Tat Lee, Sheng Zhang, Dean Carignan, Richard Edgar, Nicolo Fusi, Nicholas King, Jonathan Larson, Yuanzhi Li, Weishung Liu, Renqian Luo, Scott Mayer McKinney, Robert Osazuwa Ness, Hoifung Poon, Tao Qin, Naoto Usuyama, Chris White, and Eric Horvitz. 2023. Can generalist foundation models outcompete special-purpose tuning? case study in medicine.

OpenAI. 2024a. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/.

OpenAI. 2024b. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Sean Shi, Michael Choi, Anish Agrawal, Arnav Chopra, and et al. Adam Khoja. 2025. Humanity's last exam.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms.

Zhihong Shao, Yihong Zhang, Qingyu Wang, Yan Xu, Kai Xu, Jiang Bian, Yiming Liu, Xuanwei Liu, Peng Liu, et al. 2024. Deepseek math: Pushing the limits of mathematical reasoning in open language models.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. 2024a. MMLU-pro: A more robust and challenging multi-task language understanding benchmark. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. 2024b. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation.

Yupeng Xie, Yuyu Luo, Guoliang Li, and Nan Tang. 2024. Haichart: Human and ai paired visualization system. *Proceedings of the VLDB Endowment*, 17(11):3178–3191.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024a. Rest-mcts*: Llm self-training via process reward guided tree search.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2024b. Aflow: Automating agentic workflow generation.

Zirui Zhao, Wee Sun Lee, and David Hsu. 2023. Large language models as commonsense knowledge for large-scale task planning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*.

A Appendix

A.1 AutoSwarm Training Setup

We provide comprehensive details of our training setup and hyperparameters. The training process consists of two main components: (1) the orchestrator model training using GRPO algorithm, and (2) the workflow execution environment setup.

Orchestrator Training We use Qwen2.5-Coder-32B-Instruct as our base model for the orchestrator, which is trained using the Group-based Policy Optimization (GRPO) algorithm implemented on the OpenRLHF framework. For each input query, we sample 8 different responses to form a group for advantage estimation. The training process involves episodes of 1024 samples each. The model parameters are updated using the Adam optimizer with weight decay. The training converges at approximately 90 steps, with reward signals reaching saturation. We select the checkpoint at step 96 for our final evaluation experiments.

Workflow Execution Environment During training, we maintain a resource pool of 16 Qwen2.5-7B-Instruct models as executors. The workflow execution feedback accounts for approximately 15% of the total time in each RL training step. This overhead can be further reduced through increased parallelization or by overlapping with other RL training steps.

Training Data The RL training data consists of 3,000 problems sampled from the MATH500 dataset's training set. We carefully select problems that meet two criteria: (1) the Executor LLM must achieve at least one correct answer in 16 repeated sampling attempts with a temperature of 0.6, and (2) the accuracy rate must remain below 0.9. This ensures that the selected problems are both solvable and sufficiently challenging for the Executor LLM.

We provide the detailed hyperparameters for our training setup in Table 4.

Table 4: Training Hyperparameters

Hyper Parameter	Value
Base Model	Qwen2.5-Coder-32B-Instruct
Train Batch Size	256
Micro Train Batch Size	2
Rollout Batch Size	128
Micro Rollout Batch Size	4
Learning Rate	1e-6
Number of Training Episodes	20
Maximum Epochs	1
Prompt Max Length	2,096
Generation Max Length	2,048
Initial KL Coefficient	1e-8
Number of GPUs	64 (8 GPUs × 8 nodes)
VLLM Engine Count	16
VLLM Tensor Parallel Size	2
Advantage Estimator	GRPO
DeepSpeed ZeRO Stage	3
Mixed Precision	BF16
Random Seed	42

Infrastructure Details The training is conducted using DeepSpeed ZeRO Stage 3 for efficient memory usage and mixed precision training with BF16. We employ gradient checkpointing and Adam optimizer offloading to manage memory constraints. The distributed training setup spans across 8 nodes with 8 GPUs each, totaling 64 GPUs. For workflow execution, we utilize VLLM with 16 engines and a tensor parallel size of 2 to optimize inference throughput.

A.2 AutoSwarm Orchestrator Prompt

We present the complete orchestrator prompt used to guide the model in workflow design and execution. The prompt consists of several key components:

1. A role definition establishing the model as an expert in LLM-based workflow design
2. A comprehensive list of available specialized agent types (Generator, Fuser, Ranker, Critic, Refiner, Verifier, Summarizer, and Formatter)
3. A detailed implementation framework showing the core classes and interfaces
4. Critical requirements for workflow creation and execution

````
from agentic.model import Workflow, Agent


PROMPT_PREFIX = f"""You are an expert in designing LLM-based Agentic workflows. Your task is to
design a workflow that maximizes the synergistic potential of multiple specialized agents working
together.

# Valid Agent Types:
The system supports the following specialized agent types:
1. **Generator** - Proposes different mathematical solution approaches and strategies
2. **Fuser** - Combines multiple solution methods and integrates different mathematical perspec-
tives
3. **Ranker** - Evaluates solutions based on efficiency, elegance, and correctness
4. **Critic** - Identifies logical errors, optimization opportunities, and validates mathematical
reasoning
5. **Refiner** - Optimizes solutions by simplifying expressions and improving mathematical
clarity
6. **Verifier** - Checks mathematical proofs, validates calculations, and ensures solution
completeness
7. **Summarizer** - Creates concise explanations of mathematical solutions and key insights
8. **Formatter** - Structures mathematical expressions and equations in clear, standard notation


# Implementation Framework:
```python
import os
import asyncio
import logging
from typing import Dict, List, Literal
from openai import AsyncOpenAI
from dataclasses import dataclass

{inspect.getsource(Agent)}

{inspect.getsource(Workflow)}

def create_workflow() -> Workflow:
    # Implement your workflow here
    ...

workflow = create_workflow()
response = await workflow.run(user_input="[User Input]")
```


# Critical Requirements:
1. You may include your design rationale and explanations
2. Your implementation MUST start with '```python\\ndef create_workflow()'
3. Each agent name MUST include one of the valid agent types.
4. Provide ONLY the function implementation code

# User Input:
"""
````

Figure 5: The complete orchestrator prompt template used in AutoSwarm. The prompt includes the role definition, available agent types, implementation framework with core classes, and critical requirements for workflow creation. This structured prompt ensures consistent and effective workflow generation across different mathematical problems.