LLM-as-Scheduler: Agentic Workflow Dynamic Scheduling

Dawei Xiang[1], **Kexin Chu**[1], **Wenyan Xu**[2], **Wenhui Zhang**[3], **Wei Zhang**[1]*
[1]University of Connecticut
[2]Mohamed bin Zayed University of Artificial Intelligence
[3]Roblox
{ieb24002,kexin.chu,wei.13.zhang}@uconn.edu
Wenyan.Xu@mbzuai.ac.ae, wenhuizhang@roblox.com

Abstract

As large language models (LLMs) become more capable, many applications are shifting from a single LLM call to multi-agent systems. Manually designed or automatically optimized workflows often include multiple verification and testing stages. These stages can improve accuracy but also introduce substantial latency and increased token consumption. We find that many requests do not require such heavy-weight processing and are solvable by a single strong agent. To address this inefficiency, we propose LLM-as-Scheduler (LAS), a system that dynamically routes each query through a workflow. LAS uses a two-stage cascade: a lightweight gate that quickly checks each agent's output, and an LLM-based scheduler that makes fine-grained routing decisions using query features and gate signals. Experiments show that LAS reduces token usage by 50.5% and end-to-end latency by over 36% on average, with at most a 1.4 percentage-point drop in accuracy compared with a strong fixed workflow. The code will be public at https://github.com/YoshuaDavy/LLM-as-Scheduler

1 Introduction

As large language models (LLMs) become more powerful, many applications are shifting from a single LLM call to *multi-agent* systems. A multi-agent system (MAS) connects several specialized agents—such as planners, solvers, verifiers, critics, repair agents, and summarizers—into a workflow. These systems have shown strong performance in code generation (Wu et al., 2023), tool-augmented reasoning (Li et al., 2023a), data analysis, and information retrieval (Yao et al., 2023). Deliberative patterns such as chain-of-thought (Wei et al., 2022), self-consistency (Wang et al., 2022), self-refinement (Madaan et al., 2023), and multi-agent debate (Du et al., 2023) further increase robustness on difficult reasoning tasks (Shinn et al., 2023).

Most existing multi-agent systems use *fixed* workflows. A workflow is either hand-designed or discovered by an automated workflow-optimization framework such as ADAS (Hu et al., 2024) or AFlow (Zhang et al., 2024). At runtime, every query follows the same script of generation, verification, testing, and repair. Typical workflows include multiple iterations of critique and refinement, several validation and test stages, and sometimes explicit debate between agents. These steps improve accuracy on hard instances, but they also introduce substantial and inflexible computational overhead. Each extra stage adds latency and consumes tokens even when the first strong agent already produced an acceptable answer.

In practice, query difficulty is highly skewed. As we show in § 3, many queries do not need the full workflow. On code-generation benchmarks such as MBPP and HumanEval, a large fraction of problems are solved correctly by the first generation agent. Only a smaller fraction truly benefits from later verification and repair, and some problems remain unsolved even after all agents run. Roughly 60% of requests are solved by the first agent in the chain, an additional $\sim 20\%$ are solved only with the full deliberative workflow, and the remaining $\sim 20\%$ remain unsolved. A single, heavy workflow for all queries therefore wastes computation on easy instances and still fails on a substantial tail. This situation is similar to adaptive computation in deep networks (Teerapittayanon et al., 2016; Kaya et al., 2019), where early exiting and conditional computation reduce cost by adjusting depth to instance difficulty (Graves, 2016).

These observations motivate a simple principle: **multi-agent workflows should be dynamic**. A system should not treat all queries as equally hard. Instead, it should adapt the number and type of steps to the current instance. Some queries should exit after one or two steps with strong agents, whereas others should trigger extra verification or

*Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 12752–12763
July 2-7, 2026

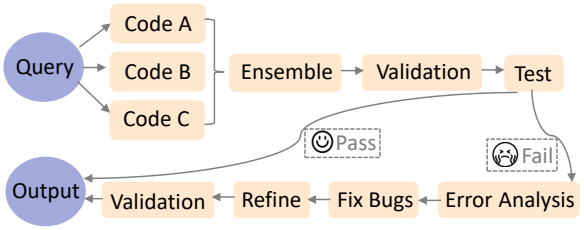

Figure 1: Example of an automatically generated multi-agent workflow in prior work (Zhang et al., 2024; Hu et al., 2024). The workflow contains multiple refinement, validation, and verification stages, which can improve accuracy on difficult instances but introduce substantial latency and computational overhead when applied uniformly to all queries.

dedicated repair agents. This requires an explicit *scheduler* or *controller* that can observe intermediate artifacts and decide how to route each query through the workflow.

To reduce overhead and avoid redundant operations on queries that do not need heavy processing, we propose a cascade scheduling system that adapts the workflow per query. We call this system **LLM-as-Scheduler (LAS)**. LAS acts as a policy layer on top of existing workflows: it observes the envelopes produced by agents and decides whether to early-exit, verify, repair, or reroute. The design follows a two-stage cascade. The first stage is a low-overhead gate unit based on scriptable checks and a small judge model. This gate runs at every step and filters out clearly bad or high-confidence cases with almost no cost. The second stage is an LLM-based scheduler that uses a larger LLM to make routing decisions when the gate detects a promising but nontrivial case.

Our evaluation results also show the effectiveness of this design. On the MBPP code-generation benchmark, LAS reduces token consumption by 63.4% and end-to-end latency by 41.9%, with at most a 0.5 percentage-point drop in accuracy compared with the baseline.

The main contributions of this paper are:

1. We identify a key limitation of current multi-agent workflows: they rely on heavy, static pipelines that apply the same long sequence of steps to every query. Many of these steps are unnecessary for most queries, which are already solved by the first strong agent.
2. We propose a *cascade scheduling system* that combines low-cost script-based checks with LLM-based routing. A lightweight gate runs at every step with negligible overhead. A larger LLM-based scheduler is invoked only when needed and operates on a compact, structured view of the workflow state.
3. We introduce a novel *LLM-as-Scheduler* design that uses a powerful LLM as the routing controller in a multi-agent system. To the best of our knowledge, this is the first work that treats the controller itself as an LLM and integrates it into a cascade over a workflow DAG.

2 Related Work

2.1 Agentic workflows and automated workflow generation

As LLMs have demonstrated strong performance across a wide range of tasks, including code generation (Austin et al., 2021; Chen, 2021), debate (Wang et al., 2023), financial analysis (Xu et al., 2025a,b), text editing (Zeng et al., 2025d,c; Xiang et al., 2025), recommendation systems (Li et al., 2023b, 2025), long-tail stabilization (Zhou et al., 2025; Zhou et al.) and multimodal reasoning (Wang et al., 2025a,b; Zeng et al., 2025a,b; Yang et al., 2026b,a), increasing attention has shifted toward agentic workflows. Agentic workflows generally fall into two complementary paradigms: single-agent and multi-agent. A single-agent system processes a task with one LLM agent and often relies on recurring design patterns to improve reliability, including chain-of-thought prompting (Wei et al., 2022), self-consistency (Wang et al., 2022), self-refinement (Madaan et al., 2023), reflexive reasoning (Shinn et al., 2023), ReAct (Yao et al., 2023), and self-critique (Madaan et al., 2023). In contrast, multi-agent and workflow-style approaches such as AutoGen (Wu et al., 2023), CAMEL (Li et al., 2023a), multi-agent debate (Du et al., 2023), and recent workflow-generation methods (Zhang et al., 2024; Hu et al., 2024) construct explicit graphs of roles and tools and execute them as predefined procedures. These workflows are typically designed and tuned by hand, which is labor-intensive and does not scale well across tasks or model families.

To reduce manual effort, several recent works aim to *automate* the design of agentic workflows. AutoFlow represents workflows as natural-language programs and iteratively optimizes them (Li et al., 2024). Other methods include AFlow (Zhang et al., 2024) and ADAS (Hu et al., 2024). These methods focus on discovering a strong workflow offline; the resulting workflow is then used as a fixed pipeline at inference time. In

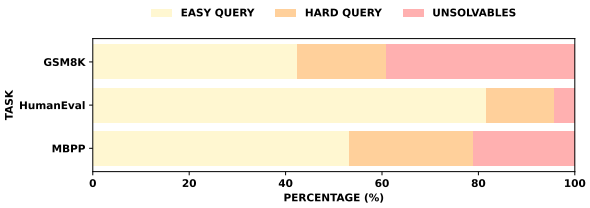

Figure 2: Portion of queries categorized as easy, hard, and unsolvable on MBPP, HUMANEVAL and GSM8K.

contrast, our work assumes that a reasonably strong workflow already exists (either manually designed or automatically generated) and studies how to *dynamically schedule* and truncate that workflow on a per-instance basis, using intermediate outputs and calibrated confidence.

2.2 Efficient multi-agent systems and cost-aware routing

Multi-agent LLM systems often incur high cost and latency due to repeated message passing, verification, and tool calls. Existing frameworks such as AutoGen make it easy to compose rich agent interactions, but in practice many workflows still follow fixed, over-provisioned scripts for every query (Wu et al., 2023). In parallel, a line of work on cost-aware routing aims to reduce token usage by learning when to use cheaper models or stop early. Examples include cascades and routing across LLM APIs (Chen et al., 2024; Dekoninck et al., 2025), as well as compilation-based approaches that tune fixed pipelines (e.g., DSPy) and automated workflow generators such as AFlow (Khattab et al., 2023; Zhang et al., 2024). Reflective methods like Self-Refine and Reflexion (Madaan et al., 2023; Shinn et al., 2023) further improve robustness via additional critique and revision steps, often at the expense of extra computation.

3 Motivation

3.1 Over-Complex Automated Workflows

Automatically generated agentic workflows (Zhang et al., 2024; Hu et al., 2024) can synthesize effective multi-agent pipelines, but the resulting designs are often overly complex. As illustrated in Fig. 1, discovered workflows frequently stack multiple rounds of refinement, validation, and verification, sometimes with nested loops. While this can help on difficult queries by increasing opportunities to detect and repair errors, deploying the same heavy pipeline for every query wastes computation: it increases latency and token cost, and provides little marginal benefit on easy cases where later stages mostly confirm the initial output.

3.2 Empirical Difficulty Distribution Analysis

To estimate the potential gains from adaptive scheduling, we measure how often a complex workflow is needed on code-generation and reasoning benchmarks. On MBPP, HUMANEVAL, and GSM8K, we run each instance through a representative automatically generated workflow (similar to Fig. 1) and record both the first-agent output (before refinement and verification) and the final workflow output (after all stages). Using the benchmark tests as an oracle, we label each query as *Easy* if the first-agent output passes, *Hard* if the first-agent output fails but the final output passes, and *Unsolvable* if both outputs fail.

Fig. 2 shows a strongly skewed difficulty distribution. Across MBPP and HUMANEVAL, roughly half of the problems are solved by the first agent alone. Only about 20% benefit from the full refinement and verification pipeline, and the remainder remain unsolved even after the workflow completes.

These results suggest two implications:

1. **Most queries do not require the full workflow.** For easy instances, additional stages increase cost and latency without improving correctness.
2. **Adaptive scheduling has substantial headroom.** If a scheduler can identify easy instances and return results earlier, it can avoid unnecessary LLM calls and tool executions, while reserving the full workflow for the smaller subset of hard instances where it helps.

4 System Design and Implementation

4.1 Cascade Scheduler

A key inefficiency in complex LLM workflows is that every query is forced through the same heavy pipeline, regardless of difficulty. We address this with an **LLM-as-Scheduler (LAS)** controller that makes per-step routing decisions based on intermediate artifacts. Given an intermediate result, LAS can either (i) *early-exit* and return the current output, or (ii) *route* the query to a more suitable next agent (e.g., refinement, verification, or testing). In this way, LAS turns a static workflow into an adaptive one.

As shown in Fig. 3, LAS is implemented as a two-stage cascade. The first stage is a low-overhead

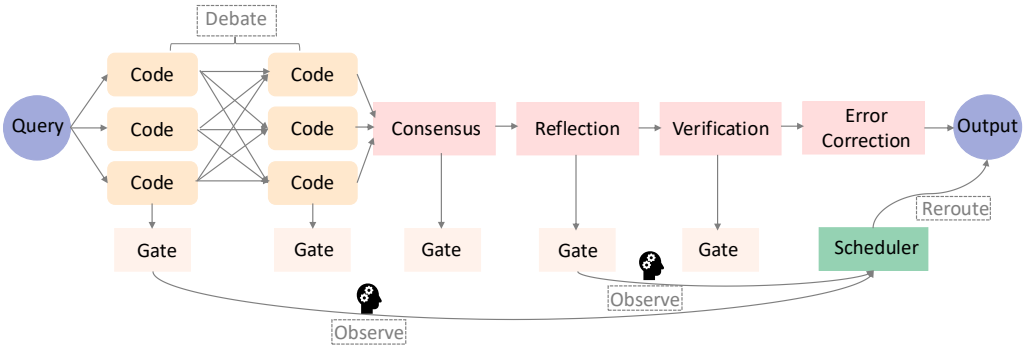

Figure 3: **LLM-as-Scheduler (LAS)**. Each agent emits a structured envelope. A fast *Cascade Gate* (low overhead) decides whether to invoke the full LAS controller. LAS fuses validator signals to decide whether to early-exit or reroute to another step/agent.

gate that quickly inspects each intermediate artifact (or a set of artifacts in debate settings). The gate routes clearly low-quality candidates to continue through the original workflow, and flags promising candidates that may be safe to skip expensive stages. The gate can be computed in parallel with workflow execution and adds negligible latency.

When a candidate is flagged, the second stage invokes a larger scheduler LLM. It combines the gate features with an LLM-judge score against a task-specific rubric, and then selects the next hop in the workflow DAG—for example, skipping refinement and going directly to testing, or terminating with an early exit.

4.2 Gate Unit

Invoking the full LAS controller after every agent would add nontrivial overhead. We therefore introduce a Cascade Gate, a lightweight decision stage that quickly determines whether a given intermediate artifact warrants detailed scheduling or can be handled by a low-cost rule.

4.2.1 Inputs and features.

The gate consumes only cheap-to-compute features:

1. **Spec adherence (binary).** Programmatic checks implemented in Python, including schema validation and regex constraints. We denote this as $f_{\text{spec}} \in \{0, 1\}$.
2. **Lite judge score.** A small evaluator (e.g., a compact encoder/classifier such as DistilBERT) produces $s_{\text{lite}} \in [0, 1]$ under a short rubric.
3. **Local agreement/consistency.** If multiple candidates are available, we compute a Jaccard similarity between the token sets of candidate outputs as an agreement score $a \in [0, 1]$.
4. **Historical agent reliability.** An exponentially weighted moving average of recent failures for the producing agent, $r \in [0, 1]$, to reflect prior reliability.

Spec adherence is used only when the task imposes an explicit output format (e.g., code generation). For tasks with unconstrained formats such as GSM8K, where the output is just a single number, we do not use spec-adherence checks.

The gate computes a cheap score

$$g = w_{\text{spec}} \cdot f_{\text{spec}} + w_{\text{lite}} \cdot s_{\text{lite}} + w_{\text{agr}} \cdot (a - 1) + w_{\text{hist}} \cdot (1 - r)$$

and compares g against a *risk-tiered* threshold $\tau_{\text{gate}}(\text{tier})$. It then returns one of:

- **Good results: pass to LAS for further consideration.** If $f_{\text{spec}} = 1$ and $g \geq \tau_{\text{gate}}$, the result appears strong and may be able to skip subsequent stages. To avoid misclassification, we forward it to LAS for fine-grained scheduling.
- **Poor results: continue with verification.** If $f_{\text{spec}} = 0$, contradictions are detected, or $g \leq \tau_{\text{gate}}$, the result is unlikely to be sufficient and may require further refinement in the original workflow.

4.2.2 Implementation

All gate features are computed with lightweight Python scripts to minimize latency and avoid additional LLM calls. For spec adherence, we rely on schema validation, regex- and type-based checks, and task-specific unit tests implemented in Python, producing a boolean indicator (or, when applicable, a calibrated score) $f_{\text{spec}} \in [0, 1]$.

The lite judge is a compact encoder-only classifier (e.g., DistilBERT) fine-tuned separately for each benchmark. During training, we run the full multi-agent workflow on all training examples. For

every agent output, we evaluate correctness and use it as a binary supervision signal. Each training instance consists of a short textual summary of the query and its candidate output, paired with the corresponding correctness label. The classifier is optimized to provide a coarse prediction of output quality. At inference time, it performs a single forward pass over the summary of a candidate output and returns a scalar confidence score $s_{\text{lite}} \in [0, 1]$.

For local agreement/consistency, we compute the average pairwise Jaccard similarity between the token sets of candidate outputs. In structured domains such as code or math, we additionally include the fraction of pairs that agree on the final answer. Both are computed using only string processing and set operations, yielding an agreement feature $a \in [0, 1]$. For other agents where there are no multiple candidate outputs, we set the a to the default value 1. We minus it by 1 in the calculation because if there is no agreement then the result is highly possible to be poor.

For parameter selection, we choose the weights $w_{\text{spec}}, w_{\text{lite}}, w_{\text{agr}}, w_{\text{hist}}$ and the gate threshold τ_{gate} via grid search to maximize performance on the validation set.

4.2.3 Low-overhead advantage

Because all of these features are obtained from pure scripts and a single small encoder, the gate is fast and low-overhead: it can run on CPU and in parallel with downstream agents, and its cost is negligible compared to a single LLM call. This makes the gate well-suited as a first-layer selector: most low-quality cases can be handled using cheap checks, while only promising cases are forwarded to the LAS scheduler for more expensive, fine-grained routing.

4.3 LLM-as-Scheduler (LAS) Routing

If the gate unit forwards the current result to LAS, we invoke a high-capacity LLM as a routing policy. The gate has already summarized the intermediate artifact into inexpensive signals (spec adherence f_{spec}, lite judge score s_{lite}, agreement a, historical reliability r, and aggregate score g), together with metadata about the current agent and step. LAS takes these gate features, a description of the workflow graph, and the current artifact, and asks a large LLM to decide how to route the query next.

Conceptually, early exit is treated as a particular routing decision: the scheduler can route to a dedicated output unit that returns the answer to the user. Otherwise, it can route to a verification or refinement agent, to a test unit (for code or math), or to another specialized agent that is better suited for the next subtask. The large LLM thus acts as a global controller that reasons over (i) what the current output claims, (ii) how strong the gate signals are, and (iii) which agents are available in the workflow, and then chooses the next hop.

In practice, we construct a structured prompt for the scheduler LLM that includes:

- a short description of the original task and risk tier,
- the current step (producing agent, its role, and what it has done),
- the gate features $(f_{\text{spec}}, s_{\text{lite}}, a, r, g)$ and any validator results,
- a catalog of available agents and units (verification, test, refinement, final output),
- the current artifact (optionally truncated or summarized for length).

The scheduler LLM is instructed to choose exactly one action from a small, discrete set of allowed routes and to return its decision in a machine-readable format that can be parsed by the orchestration layer. The orchestration layer parses the returned JSON, reads the `action` and `target` fields, and invokes the corresponding agent or output unit. For example, an action `early_exit` with target `output_unit` routes directly to the output stage; an action `test` with target `test_runner_agent` passes the artifact to the test agent; and an action `refinement` sends the artifact to a refinement agent that attempts to improve the answer before returning to verification or testing.

We implement the scheduler as a stateless function that constructs the prompt, calls the large LLM, and post-processes its response. A high-level pseudocode sketch is given below.

The function `BuildSchedulerContext` collects the task description, current agent, gate features, validator results, and a compact representation of the remaining workflow. `FormatSchedulerPrompt` serializes this context into the structured prompt shown above. `CallLargeLLM` makes a single call to the large model. `ParseSchedulerResponse` enforces a strict schema on the returned JSON and can reject malformed answers. The additional post-processing in the last two blocks implements simple policy rules, such as biasing against early exit on high-risk tiers when the gate score is not

Algorithm 1 LAS routing with an LLM scheduler

Require: Envelope E with fields: `artifact`, `gate_features`, `validators`, `meta`
Require: Workflow description W (agents, roles, edges)
Require: Risk tier t
Ensure: Action a and target unit u

```
context ← BuildSchedulerContext(E, W, t)
prompt ← FormatSchedulerPrompt(context)
response ← CallLargeLLM(prompt)
(a, u, r) ← ParseSchedulerResponse(response)
if a not in {early_exit, verification, test, refinement} then
    (a, u) ← DefaultFallbackRoute(E, W)
if t is high risk and a = early_exit and gate_features.g is modest then
    (a, u) ← OverrideToVerification(W)
return (a, u)
```

	w_{spec}	w_{lite}	w_{agr}	w_{hist}	τ_{gate}
MBPP	0.06	0.85	0.07	0.02	0.803
HumanEval	0.07	0.82	0.07	0.04	0.824
GSM8K	0	0.75	0.14	0.11	0.852

Table 1: Gate unit weights and score thresholds on different datasets.

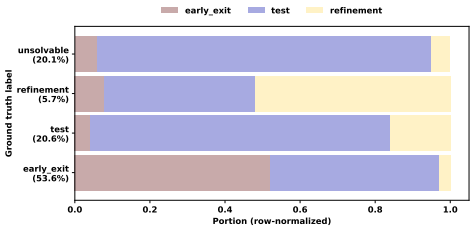

Figure 4: LAS prediction diagnostics (stacked bar chart).

very high.

5 Evaluation

5.1 Experimental setup

5.1.1 Datasets

We evaluate our method on three public benchmarks. For code generation, we use HumanEval (Chen, 2021) and MBPP (Austin et al., 2021). For mathematical reasoning, we use GSM8K (Cobbe et al., 2021). For all benchmarks, we use the official test sets only for final evaluation. We use the training sets of each benchmark to train the gate-unit judge model, and we use the validation sets for hyperparameter selection.

5.1.2 Metrics

For GSM8K, we report *Solve Rate* (%), defined as the fraction of problems whose final answer exactly matches the reference after standard normalization. For HumanEval and MBPP, we report pass@1, defined as the fraction of problems for which the single generated program passes all reference unit tests. Unless otherwise noted, we report all metrics on the official test sets after average on 5 runs. If a benchmark has no validation set, we split training set by 8:2 for the validation set. We also report output token consumption as `Tokens` and total end-to-end latency as `Lat` in the results table. We report output-token usage because, under many pricing schemes, output tokens are more expensive than input tokens and thus dominate cost.

5.1.3 Baselines

We compare LAS against six baselines: IO (single-call), Chain-of-Thought (CoT) (Wei et al., 2022), Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), ADAS (Hu et al., 2024), and AFlow (Zhang et al., 2024). These baselines cover common hand-crafted multi-step prompting strategies (CoT, Self-Refine, Reflexion) and strong automatically generated multi-agent workflows (ADAS, AFlow). AFlow is a state-of-the-art automated workflow optimization method that searches over multi-agent workflows and then executes the discovered workflow as a fixed pipeline at inference time. It can achieve high accuracy but also incurs substantial overhead.

We use AFlow as the primary reference point for comparison. In all cases, we use the same base models for generation wherever applicable, so that differences reflect the workflow and scheduling strategy rather than changes in backbone model capability. Since our main contribution is dynamic workflow adjustment, we instantiate LAS on top of the AFlow workflow, which is the most accurate but also the most expensive configuration. The cascade scheduler observes intermediate results and routes each query to appropriate downstream steps, instead of always executing the full workflow.

5.1.4 Implementation details

LAS uses a two-stage cascade for scheduling. The first stage is a lightweight gate, where we use DistilBERT as the lite judge model. The second stage invokes an LLM scheduler: we use Doubao-seed-1.6 (Doubao) as the scheduler LLM, and Doubao-seed-1.6-lite as the main executor for agents in the workflow. These models are accessed through public APIs provided by the Volcano Engine platform (volcengine). We select these models because they offer a favorable cost–quality trade-off. For

Method	MBPP			HumanEval			GSM8K		
	Acc.	Tokens	Lat. (s)	Acc.	Tokens	Lat. (s)	Acc.	Tokens	Lat. (s)
IO	79.3	1005.4	19.54	92.6	1424.7	23.52	91.7	1342.5	25.54
COT	84.2	4058.7	67.04	93.7	2678.5	87.89	94.1	6852.1	94.24
Self Refine	82.1	3804.4	79.60	92.2	6352.1	93.85	92.1	8825.4	115.14
Reflexion	85.5	4361.5	89.72	93.1	7263.5	107.20	92.9	11250.4	129.5
ADAS	89.2	5824.3	121.25	95.3	8982.4	169.64	92.3	15723.7	167.27
AFLOW(baseline)	**94.2**	6643.2	134.28	**95.9**	9390.8	171.56	**95.9**	17582.5	198.26
Ours	93.7	**2430.3**	**77.99**	94.5	**5876.5**	**108.34**	95.1	**8624.1**	**125.15**
Δ Ours vs baseline	**-0.5%**	**-63.4%**	**-41.9%**	**-1.4%**	**-37.4%**	**-36.8%**	**-0.8%**	**-50.9%**	**-36.9%**

Table 2: Comparison of methods across MBPP, HumanEval, and GSM8K benchmarks. *Acc, Tokens, Lat* refer to output accuracy, output token consumption, end-to-end latency respectively.

fairness, all baselines use the same executor models; only the workflow structure and the presence or absence of LAS differ across methods.

Gate parameter selection. We tune the gate weights $w_{\text{spec}}, w_{\text{lite}}, w_{\text{agr}}, w_{\text{hist}}$ and the gate threshold τ_{gate} via grid search on the validation set, and then report final performance on the test set. The selected values are shown in Table 1. We do not apply spec-adherence checks on GSM8K because the final output is a single number; accordingly, $w_{\text{spec}} = 0$ for GSM8K. For GSM8K, we additionally require each agent to output a short rationale together with its answer, and we use the rationale (together with the answer) as input to the lite judge model to estimate answer credibility.

Scheduler prediction diagnostics. To better understand routing behavior, we record LAS decisions and summarize them by ground-truth difficulty labels. In addition to the heatmap analysis (Fig. 6), we provide a stacked bar chart in Fig. 4.

5.2 Results and analysis

Table 2 compares our cascade scheduler with a single-agent baseline (IO), manually designed multi-step methods (CoT, Self-Refine, Reflexion), the ADAS workflow, and a complex multi-agent workflow baseline (AFLOW). Relative to other multi-step or multi-agent baselines, our method achieves higher or comparable accuracy while using substantially fewer tokens and lower latency across all three benchmarks. On MBPP and GSM8K, for instance, it improves accuracy by several percentage points over CoT, Self-Refine, and Reflexion while using roughly 2–3$\times$ fewer tokens and notably less latency, suggesting that dynamic routing offers a better accuracy–efficiency trade-off than fixed pipelines.

Compared with AFLOW (the most accurate but also the most expensive configuration), our scheduler trades a small accuracy decrease for large efficiency gains. As shown in the Δ row of Table 2, accuracy drops by at most 1.4 percentage points, while output-token usage decreases by about 43–63% and latency by about 36–41%.

5.3 Ablation Study

We ablate the gate and LAS components on MBPP (Table 3).

Only Gate disables LAS and uses the gate score as the sole early-exit signal. It is faster and more token-efficient than the AFLOW baseline because it skips many downstream verification steps, but it suffers a large accuracy drop (see also § 5.5).

Only LAS removes the gate and lets LAS schedule on every agent output. It largely preserves accuracy relative to fixed AFLOW, but substantially increases token usage. Since LAS runs asynchronously with the main workflow, the latency increase is limited.

Our full *Gate+LAS* cascade matches *Only LAS* closely in accuracy while keeping tokens and latency low, providing a better quality–cost balance in practice.

Method	Acc. (%)	Tokens	Lat. (s)
AFLOW (baseline)	**94.2**	6643.2	134.28
Only Gate	86.7	2516.4	**65.24**
Only LAS	93.9	3678.1	90.25
Gate+LAS (ours)	93.7	**2430.2**	77.99

Table 3: Ablation of the gate and LAS components on MBPP. For *Only Gate* and *Gate+LAS*, we tune the gate threshold on the validation set and report results on the test set.

Ablation on gate unit components. As shown in Table 4, neither the lite judge alone nor the combination of the other three components matches the performance of the full gate unit. This indicates that all components make complementary contri-

butions, and that their joint use leads to the best overall performance.

Component	Precision	Accuracy	Recall
Spec adherence + agreement + history + lite judge	66.29%	70.64%	98.60%
Spec adherence + agreement + history	61.90%	64.40%	95.78%
Lite judge	63.50%	66.20%	94.20%

Table 4: Ablation study on different components of the gate unit.

5.4 Overhead analysis

Table 5 breaks down the overhead of the gate and LAS scheduler. The gate runs at every step, is script-based, and makes no additional generative LLM calls; it uses zero extra tokens and adds only 0.15 s per invocation. Relative to a typical workflow agent (1106 tokens and 26.78 s), its overhead is negligible.

While the LAS module incurs latency and token consumption similar to those of an average agent, it can route workflows to early-exit or test stages, thereby substantially reducing computation in later steps. Consequently, LAS achieves a significant reduction in end-to-end latency, as shown in Table 2.

Component	Tokens	Lat. (s)
Average workflow agent	1106	26.78
Gate unit	0.0	0.15
LAS scheduler	725	19.44

Table 5: Per-component overhead of the gate and LAS scheduler on MBPP.

5.5 Prediction accuracy analysis

Gate-only protocol. In the `Only Gate` setting, the gate assigns a confidence score to each intermediate artifact. We sweep a threshold τ: if the score is at least τ, we early-exit and return the current output; otherwise, we proceed to the next stage.

5.5.1 Gate unit

We evaluate the gate on MBPP using the protocol in § 5.5. Treating an output as acceptable if it early-exits at threshold τ, we compare predictions against unit-test correctness and sweep τ to obtain an ROC curve. As shown in Fig. 5 and Table 6, the gate reaches 70.64% accuracy at its best operating point. Recall is high (98.60%), so the gate rarely misses correct outputs; precision is lower (66.29%), so some outputs flagged as safe are incorrect. This supports using the gate as a trigger, with LAS handling borderline cases to reduce unsafe early exits.

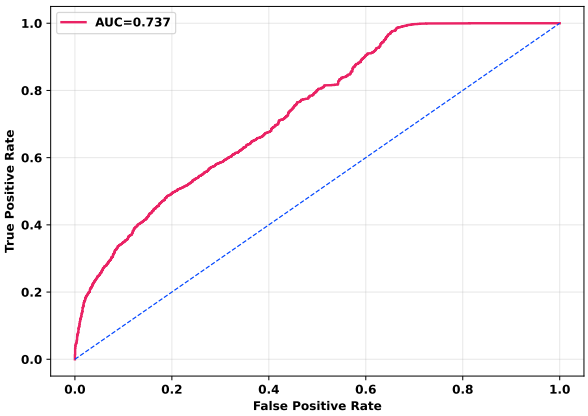

Figure 5: ROC curve for the gate unit predicting output correctness on MBPP.

Model	Accuracy	Precision	Recall
Gate	70.64%	66.29%	98.60%

Table 6: Best operating point of the gate unit on MBPP (chosen by maximizing accuracy).

5.5.2 LAS routing accuracy

We evaluate LAS by assigning each agent output a ground-truth routing label based on the earliest stage at which the instance can be solved: early exit (already passes tests), test (fails currently but passes after verification), refinement (requires downstream refinement), or unsolvable (fails after all stages). We compare LAS actions to these labels. Fig. 6 and Fig. 4 show that LAS often matches the optimal routing choice, with an average routing accuracy of 65.72%. Many errors are benign (e.g., predicting `Test` instead of `Early exit`), which adds limited verification cost while still avoiding expensive refinement. Harmful errors—where the ground truth is `Refinement` but LAS predicts `Early exit` or `Test`—occur in only 2.7%, consistent with the small end-to-end accuracy drop.

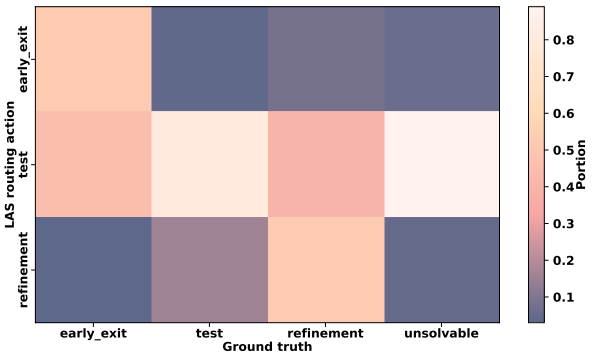

Figure 6: Heatmap of LAS routing decisions versus ground-truth routing labels.

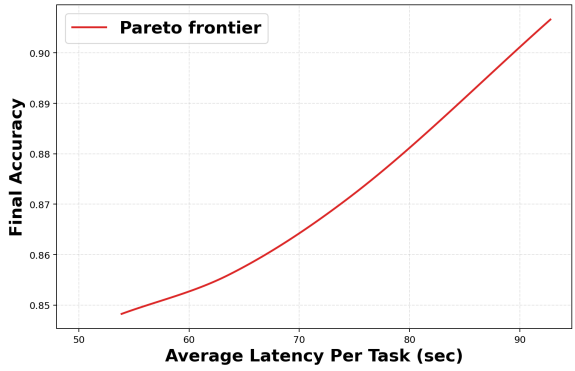

Figure 7: Pareto frontier of `Only Gate`.

5.6 Overhead and accuracy trade-off

We plot the Pareto frontier (accuracy vs. end-to-end latency) for `Only Gate`. We use the threshold-sweep protocol in § 5.5 to obtain operating points.

Fig. 7 shows the trade-off between accuracy and overhead: if the threshold decreases, more cases will early exit, the end-to-end latency will be lower, but the accuracy will also decrease.

The training overhead for the lite-judge(DistillBERT) is also low. Only 50 epochs of training can achieve the accuracy of 70.64% on the MBPP test set. And the total training time is less than one hour.

6 Acknowledgment

This work is supported in part by the National Science Foundation under Award # 2530909 and by NEUTC.

7 Conclusion

We highlight the inefficiency of fixed multi-agent workflows and propose LLM-as-Scheduler (LAS), a cascade controller that adapts execution per query via a lightweight gate and an LLM scheduler. Experiments show that LAS preserves nearly all accuracy while reducing token usage by 50.5% and latency by over 36%.

8 Limitations and Future Work

Limitations. Our approach relies on a lightweight decision component (a small language model) to make fast routing and early-exit determinations. While this design is critical for reducing end-to-end cost and latency, it may introduce errors when the small model is poorly calibrated for a new domain. In addition, our current evaluation focuses on a limited set of benchmarks and workflow configurations, and the generalizability of the learned gating/scheduling behavior across tasks, model families, and tooling environments remains under study.

Future Work. Our next step is to improve robustness and transfer. We plan to investigate (i) more transferable decision modules that reduce reliance on benchmark-specific supervision (e.g., domain-agnostic confidence signals and self-supervised calibration), (ii) cross-task and cross-workflow evaluation protocols to better characterize when LAS transfers without retraining, and (iii) extend our framework to boarder benchmarks including RAG and Tool-using tasks. We also aim to explore replacing the small model with alternative low-cost decision mechanisms and to analyze the trade-off between decision accuracy and scheduling overhead under varying latency and budget constraints.

9 AI assistant use explanation

We used AI assistant in improving writing and writing code in experiments.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Lingjiao Chen, Matei Zaharia, and James Zou. 2024. Frugalgpt: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Jasper Dekoninck, Maximilian Baader, and Martin Vechev. 2025. A unified approach to routing and cascading for llms. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 12987–13010. PMLR.

Doubao. Doubao-seed-1-6. `https://www.aibase.com/tool/358373`.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2023. Improving

factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*.

Alex Graves. 2016. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *The Thirteenth International Conference on Learning Representations*.

Yigitcan Kaya, Sanghyun Hong, and Tudor Dumiţraş. 2019. Shallow-deep networks: Understanding and mitigating network overthinking. In *International Conference on Machine Learning (ICML)*.

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

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023a. Camel: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yuyuan Li, Chaochao Chen, Yizhao Zhang, Weiming Liu, Lingjuan Lyu, Xiaolin Zheng, Dan Meng, and Jun Wang. 2023b. Ultrare: Enhancing receraser for recommendation unlearning via error decomposition. *Advances in Neural Information Processing Systems*, 36:12611–12625.

Yuyuan Li, Yizhao Zhang, Weiming Liu, Xiaohua Feng, Zhongxuan Han, Chaochao Chen, and Chenggang Yan. 2025. Multi-objective unlearning in recommender systems via preference guided pareto exploration. *IEEE Transactions on Services Computing*.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. 2024. Autoflow: Automated workflow generation for large language model agents. *CoRR*, abs/2407.12821.

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.

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.

Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2016. Branchynet: Fast inference via early exiting from deep neural networks. In *International Conference on Pattern Recognition (ICPR)*.

volcengine. Ark model lists. https://www.volcengine.com/docs/82379/1330310?lang=zh.

Boshi Wang, Xiang Yue, and Huan Sun. 2023. Can chatgpt defend its belief in truth? evaluating llm reasoning via debate. In *Findings of the association for computational linguistics: EMNLP 2023*, pages 11865–11881.

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

Zixuan Wang, Jinghao Shi, Hanzhong Liang, Xiang Shen, Vera Wen, Zhiqian Chen, Yifan Wu, Zhixin Zhang, and Hongyu Xiong. 2025a. Filter-and-refine: A mllm based cascade system for industrial-scale video content moderation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pages 873–880.

Zixuan Wang, Yu Sun, Hongwei Wang, Baoyu Jing, Xiang Shen, Xin Luna Dong, Zhuolin Hao, Hongyu Xiong, and Yang Song. 2025b. Reasoning-enhanced domain-adaptive pretraining of multimodal large language models for short video content governance. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1104–1112.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

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. *arXiv preprint arXiv:2308.08155*.

Dawei Xiang, Wenyan Xu, Kexin Chu, Tianqi Ding, Zixu Shen, Yiming Zeng, Jianchang Su, and Wei Zhang. 2025. Promptsculptor: Multi-agent based text-to-image prompt optimization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 774–786.

Wenyan Xu, Dawei Xiang, Yue Liu, Xiyu Wang, Yanxiang Ma, Liang Zhang, Shu Hu, Chang Xu, and Jiaheng Zhang. 2025a. Finmultitime: A four-modal bilingual dataset for financial time-series analysis. *arXiv preprint arXiv:2506.05019*.

Wenyan Xu, Dawei Xiang, Rundong Wang, Yonghong Hu, Liang Zhang, Jiayu Chen, and Zhonghua Lu. 2025b. Learning explainable stock predictions with tweets using mixture of experts. *arXiv preprint arXiv:2507.20535*.

Panqi Yang, Haodong Jing, Nanning Zheng, and Yongqiang Ma. 2026a. Instrucrobo: Object-centric multi-instruction decoupling model for explainable robotic manipulation. *Engineering Applications of Artificial Intelligence*, 171:114166.

Panqi Yang, Haodong Jing, Nanning Zheng, and Yongqiang Ma. 2026b. Unihoi: Unified human-object interaction understanding via unified token space. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 11640–11648.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Shuang Zeng, Xinyuan Chang, Mengwei Xie, Xinran Liu, Yifan Bai, Zheng Pan, Mu Xu, and Xing Wei. 2025a. Futuresightdrive: Thinking visually with spatio-temporal cot for autonomous driving. *arXiv preprint arXiv:2505.17685*.

Shuang Zeng, Dekang Qi, Xinyuan Chang, Feng Xiong, Shichao Xie, Xiaolong Wu, Shiyi Liang, Mu Xu, and Xing Wei. 2025b. Janusvln: Decoupling semantics and spatiality with dual implicit memory for vision-language navigation. *arXiv preprint arXiv:2509.22548*.

Yiming Zeng, Jinghan Cao, Zexin Li, Wanhao Yu, Zhankai Ye, Dawei Xiang, Ting Hua, Xin Liu, Shangqian Gao, and Tingting Yu. 2025c. Hyperedit: Unlocking instruction-based text editing in llms via hypernetworks. *arXiv preprint arXiv:2512.12544*.

Yiming Zeng, Wanhao Yu, Zexin Li, Tao Ren, Yu Ma, Jinghan Cao, Xiyan Chen, and Tingting Yu. 2025d. Bridging the editing gap in llms: Fineedit for precise and targeted text modifications. *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 2193–2206.

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*.

Xiaoling Zhou, Ou Wu, and Nan Yang. 2025. Class and attribute-aware logit adjustment for generalized long-tail learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 22991–22999.

Xiaoling Zhou, Mingjie Zhang, Zhemg Lee, Yuncheng Hua, Wei Ye, Flora D Salim, Shikun Zhang, and 1 others. Boosting resilience of large language models through causality-driven robust optimization. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

A Example of LAS system prompt

An example prompt for the scheduler LLM is as follows:

SYSTEM:
You are a routing controller for a multi-agent LLM workflow. Given the current intermediate result, gate signals, and the available agents, decide where to send the request next. Be conservative on high-risk tasks.
USER:
[Task]
You are solving a programming problem. The final goal is to produce a correct Python function that passes all tests.
[Risk tier]
tier = "medium"
[Current step]
agent_name: "draft_coder"
agent_role: "Generate an initial solution."
"step_id: 3
[Gate features]
spec_adherence: 1
lite_judge_score: 0.82
agreement_score: 0.76
historical_reliability: 0.90
gate_score_g: 0.84
[Validator summary]
* syntax_check: passed
* basic_unit_tests: not_run
* safety_filters: passed
[Available routes]
1. early_exit_to: "final_output_unit"
2. continue send_to: [next_agent]
3. verification send_to: "test_runner_agent"
[Current artifact] <snippet of the generated code here>
Decision:
Choose the single best action from the list above.
Respond in strict JSON with fields:
{"action": "<one of: early_exit, verification, test, refinement>",
"target": "<name of the target agent or unit>",
"reason": "<short natural language justification>"}.