SkyAPI: Cost-Effective API Orchestration for Compound LLM Serving

Anonymous ACL submission

Abstract

The rapid adoption of large language models (LLMs) has accelerated a shift from single-request inference toward *compound* and *agentic* applications, where multiple LLM calls collaborate through structured, graph-based workflows. These systems increasingly rely on third-party inference APIs, whose latency and pricing vary widely across providers. However, existing API routing approaches optimize requests in isolation, overlooking inter-request dependencies, which substantially degrades cost effectiveness in compound execution.

We present SkyAPI, a structure-aware API orchestration framework for compound LLM serving. SkyAPI optimizes provider selection for compound execution graphs by decomposing dynamic workflows into ordered execution stages that explicitly capture dependencies and straggler effects. At each stage, SkyAPI formulates provider selection as an integer linear program that jointly optimizes stage makespan and monetary cost, while incrementally compensating for execution-time uncertainty (e.g., unknown request dependencies and response lengths) and runtime variation (e.g., API latency fluctuations). SkyAPI further incorporates a prefix-warmup scheduling mechanism that maximizes cache reuse without increasing serving latency. Extensive real-world evaluation across DeepResearch, GAMA-Bench, and ten commercial API providers shows that SkyAPI achieves 54% cost savings and 62% latency reduction comparing to existing baselines.

1 Introduction

With rapid advances in capability and adoption, large language models (LLMs) have driven a shift from single LLM calls toward *compound LLM serving* systems, in which multiple LLM invocations collaborate to solve complex, multi-step tasks through structured interaction patterns such as task planning, decomposition, and aggregation, forming increasingly complex execution graphs. These compound applications, including multi-agent systems (MAS) and inference-time scaling workloads, often rely heavily on third-party LLM APIs to execute individual requests. However, as workflows grow in depth and width, naive API usage quickly leads to substantially higher end-to-end latency and monetary cost (e.g., 3–34× compared to a single LLM request) (Gao et al., 2025).

Increasingly, balancing responsiveness and monetary costs in API selection becomes a fundamental challenge. With the democratization of LLMs and the growing availability of capable public models, many providers now offer API access to the same underlying models deployed on their own infrastructure. Recent efforts aggregate such APIs behind unified interfaces to simplify API management and user specification. For example, OpenRouter (ope, b) supports more than 65 API providers, with 17 providers hosting APIs for Qwen3-235B-A22B access. Even when serving the same model, we notice that these providers exhibit 5-15× differences in latency, throughput, and pricing, making API provider selection indispensable (§2).

Unfortunately, the *structurally dependent* nature of compound LLM serving lies beyond the design envelope of existing API selection (routing) approaches (e.g., OpenRouter, SkyLLM (Zhao and Zhu, 2025)), which primarily optimize *isolated* requests. First, because requests are interdependent, accelerating a request that is not on the workflow's latency-critical path does not reduce end-to-end latency, yet may significantly increase costs when selecting a faster yet more expensive API provider. Second, dependencies induce *context overlap*. For example, calls may share common prompt prefixes (e.g., in inference-time scaling), creating opportunities for cross-request prefix reuse that per-request routing fails to exploit. Third, compound requests typically execute over longer time horizons, dur-

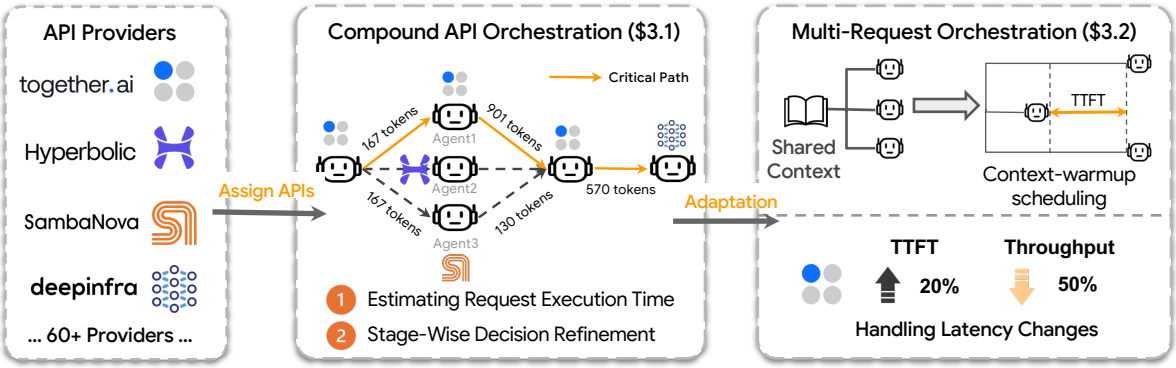

Figure 1: SkyAPI determines API assignments and their orchestration for compound LLM serving under uncertainties (e.g., inter-request dependencies), while accounting for shared context to improve latency–cost trade-offs.

ing which provider performance can fluctuate due to load and network dynamics, leading to unpredictable latency propagating to downstream calls.

This paper introduces SkyAPI, a structure-aware API selection framework for optimizing the cost-effectiveness of compound LLM serving (Figure 1). SkyAPI accounts for execution-time uncertainties, including unpredictable response lengths, inter-request dependencies, and latency fluctuations, to dynamically select API providers for a given model. By optimizing provider selection at runtime without altering the underlying model (thus lossless), SkyAPI minimizes end-to-end serving latency and monetary cost for compound workflows, going beyond existing per-request API selection advances.

This paper makes the following contributions:

- We characterize the latency–cost trade-offs in compound LLM serving through extensive real-world measurements across tens of commercial API providers (§2).
- We propose SkyAPI, the first structure-aware API orchestration framework that schedules LLM requests under runtime uncertainties (§3).
- We propose a novel stage-level API orchestration algorithm to incrementally optimize the global cost-effectiveness;
- We demonstrate the effectiveness of SkyAPI across diverse compound LLM benchmarks using commercial LLM services (§4).

2 Background and Motivation

We begin with an overview of compound LLM serving and our problem formulation (§2.1), followed by our characterization of commercial API services and a discussion of the limitations of existing approaches that motivate our work (§2.2).

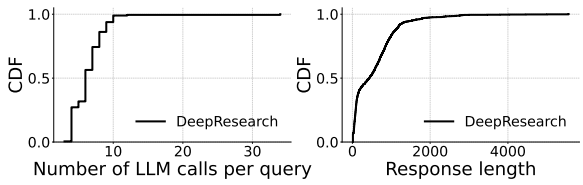

Figure 2: LLM serving is increasingly compound, involving uncertain calls to finish a task.

2.1 Problem Formulation

The LLM serving paradigm has shifted from single requests toward *compound* execution, where multiple LLM calls collaborate to complete a task. Multi-agent frameworks such as MetaGPT (Hong et al., 2023), AutoGen (Wu et al., 2024), and ChatDev (Qian et al., 2024) orchestrate LLM agents with different roles to solve complex problems, such as end-to-end software development. Similarly, as shown in Figure 2, inference-time scaling techniques and deep research applications explore multiple reasoning trajectories and aggregate their outputs to improve solution quality, with (sub)calls in a compound request of varying lengths. Increasingly, the orchestration of LLM requests forms an execution graph and evolves dynamically; for example, a planner agent may decide which worker agents to invoke based on information gathered from previous interaction rounds (Li et al., 2025).

We model the execution of a compound LLM serving request as a directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$[1], where each node $v \in \mathcal{V}$ represents an LLM invocation and each directed edge $(u, v) \in \mathcal{E}$ denotes the execution dependency, indicating that v's execution needs u's completion. For a user-specified model $m \in \mathcal{M}$, the API orchestration

[1]Requests with execution loops (e.g., self-reflection) can be unrolled into DAGs.

system must select a provider $p \in \mathcal{P}$ that offers an API serving the same model, thus not altering the response (i.e., lossless). This selection determines both the expected latency $L_{v,m,p}$ and the monetary cost $C_{v,m,p}$ of executing node v. Both quantities depend on the request's input length l_{in} and (often unknown) output length l_{out}.

Following common service objectives (ope, b; Zhao and Zhu, 2025), our goal is to optimize the trade-off between end-to-end serving latency and monetary cost, controlled by a parameter $\alpha \geq 0$. Formally, we seek an API orchestration P^* that minimizes the following objective ($OPT(\mathcal{G})$):

$$\min \; Latency(\mathcal{G}) + \alpha \cdot Cost(\mathcal{G}) \qquad (1)$$

where $Latency(\mathcal{G})$ is determined by the length of the critical path in the execution DAG, and $Cost(\mathcal{G})$ is the sum of costs incurred by all LLM invocations.

2.2 Characterizing LLM APIs

The rapid proliferation of competitive open-source models has given rise to a rich and rapidly growing ecosystem of independent API providers. As of January 2026, OpenRouter (ope, b) offers unified API access to more than 60 model providers, including 14 providers hosting DeepSeek-V3 and 17 providers hosting Qwen3-235B-A22B.

Although these providers serve identical models with the same theoretical capabilities, our real-world measurements across ten DeepSeek API providers reveal substantial variation in cost effectiveness. To quantify this, we utilized MMLU-Pro (Wang et al., 2024) benchmark to periodically dispatch requests to these API providers over a 24-hour period to record their serving latency. These measurements reveal substantial variation in cost-effectiveness across different endpoints.

Latency, throughput, and cost vary across API providers. The latency of a single request can be decomposed into time-to-first-token (TTFT) plus time-between-tokens (TBT) multiplied by the number of generated tokens. TTFT is primarily dominated by network overhead and prefilling computation, whereas TBT reflects the provider's decoding throughput. As shown in Figure 3, Throughput exhibits substantial variability across providers, with the 95th-percentile throughput ranging from *28* tokens/s to *155* tokens/s. We also observe significant disparities in TTFT and per-token pricing. These results highlight the non-trivial performance–cost trade-offs faced by users when selecting API providers.

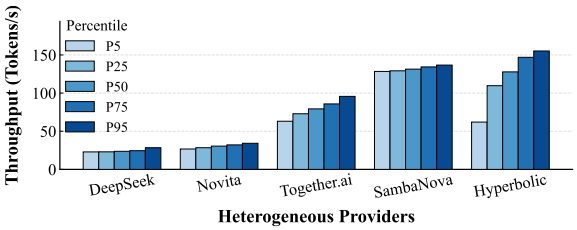

Figure 3: Serving latency varies across API providers.

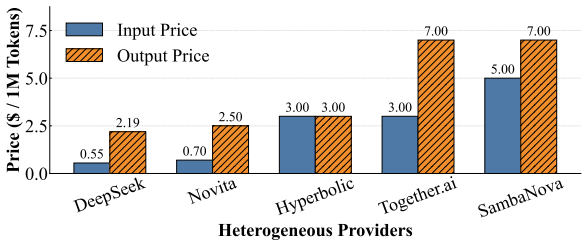

Figure 4: Per-token serving cost varies across providers.

TTFT and throughput performance vary over time. We monitor TTFT and decoding throughput from Together.ai over a two-hour interval. As shown in Figure 5, TTFT remains largely stable throughout the measurement period. In contrast, decoding throughput exhibits substantial temporal variability, fluctuating within a wide dynamic range, with the maximum token generation rate reaching nearly $3\times$ the minimum observed speed. These pronounced throughput fluctuations highlight the importance of actively tracking the real-time performance of each provider to sustain optimal routing and orchestration decisions.

Limitations of Existing Solutions. Despite the growing availability of heterogeneous API providers and increasingly complex compound serving workflows, existing API routing solutions (ope, a; Zhao and Zhu, 2025) mostly optimize single-request metrics, such as selecting the cheapest provider for each request independently. They often over-prioritize requests that are not on the critical path, leading to substantial degradation in cost effectiveness. Moreover, inter-request dependencies often induce shared context (e.g., common prompt prefixes), creating opportunities for cross-request context reuse that per-request routing fails to exploit. Finally, effective API orchestration in compound workflows must operate under dynamic and partially observable execution graphs, a challenge that existing solutions fail to address.

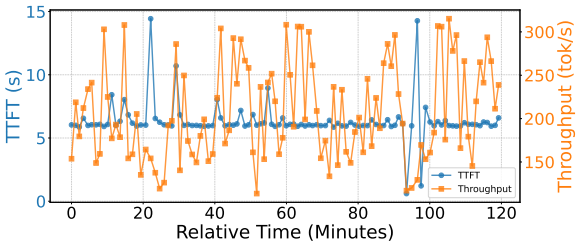

Figure 5: Practical API performance varies over time.

3 SkyAPI Design

This paper introduces SkyAPI to optimize the trade-off between serving latency and monetary cost by minimizing $Latency(\mathcal{G}) + \alpha \cdot Cost(\mathcal{G})$ for compound LLM serving requests (§2.1). We next describe how SkyAPI addresses unpredictable response lengths and inter-request dependencies (§3.1), as well as shared context (e.g., prefix reuse) and latency fluctuations across requests (§3.2).

3.1 Resilient Compound API Orchestration

Cost-effective API usage for compound LLM serving requires more than existing advances, which select providers for individual requests in isolation. Instead, it requires aligning provider performance with the structural constraints of the execution workflow, i.e., orchestrating API choices across interdependent LLM calls. For example, in a parallel execution stage, the stage completion time is determined by the slowest request; therefore, requests that are not on the critical path can be assigned to slower but cheaper API providers without increasing overall stage latency.

Optimizing the DAG-level cost effectiveness requires addressing two key sources of uncertainty: (1) the execution time of individual requests, and (2) their execution dependencies.

Estimating Request Execution Time. Estimating the execution time of individual requests is essential for identifying the critical path of a compound workflow. The execution time of a request r depends on its generation length (i.e., the number of input tokens l_r^{in} and output tokens l_r^{out}), as well as the provider-specific processing speeds for prefilling and decoding. Let T_m^{in} and T_m^{out} denote the per-token prefilling and decoding latencies of provider p, respectively. Since practical per-token processing speeds are agnostic to request content (§2.2), we omit the dependence on r. The estimated latency L_r^p and monetary cost C_r^p of executing request r on provider p are then given by:

$$L_r^p = T_p^{\text{in}} \cdot l_r^{\text{in}} + T_p^{\text{out}} \cdot l_r^{\text{out}},$$
$$C_r^p = C_p^{\text{in}} \cdot l_r^{\text{in}} + C_p^{\text{out}} \cdot l_r^{\text{out}}.$$

Here, C_p^{in} and C_p^{out} denote the provider-specified per-token costs for input and output tokens. The processing latencies T_p^{in} and T_p^{out} are provider-dependent and are estimated from recent request observations, with online updates to account for runtime fluctuations (§3.2).

The remaining challenge is to estimate the unknown output length l_p^{out}. Following recent advances in LLM serving and scheduling (Zhang et al., 2026), we employ a small language model (SLM), such as Bert, as a lightweight predictor to estimate the expected output length conditioned on the input prompt. Our SLM achieves sufficient prediction accuracy while introducing negligible overhead (less than 2%; §4.2). More importantly, we describe next how SkyAPI remains robust to prediction errors during scheduling.

Resilient Graph-level Orchestration. Orchestrating API providers to optimize the cost effectiveness of an execution DAG is challenging due to inter-request dependencies, even when accurate output-length predictions are available. In compound workflows, downstream requests often consume the outputs of upstream requests as context. For example, a planning agent may branch execution into different subtasks, a filtering agent may prune execution paths, and an open-ended reasoning agent may trigger additional requests based on intermediate results. Therefore, the input length of downstream requests, and consequently their response length and cost, can hardly be predicted.

Even worse, the execution DAG may be partially observable at the start of execution and evolves dynamically as requests complete and new dependencies emerge. These properties make it impractical to compute a globally optimal API orchestration plan ahead of time, necessitating a resilient, online approach to graph-level orchestration.

Our key insight is to decompose the intractable graph-level optimization problem into observable, stage-level orchestration, and to iteratively refine orchestration decisions across stages to optimize global cost effectiveness. Concretely, given the realized execution DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we partition the node set $\mathcal{V}$ into an ordered sequence of execution stages $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$, where each

stage $S_k \subseteq \mathcal{V}$ contains requests whose execution horizons are mutually independent. Within a stage, requests can be invoked and executed in parallel; their outputs are subsequently aggregated or consumed by the downstream stage S_{k+1}.

Beyond circumventing the infeasibility of predicting the entire execution graph upfront, stage-level decomposition offers several practical advantages. First, it generalizes naturally to various compound LLM serving paradigms (DAGs). Note that chain-based execution can be viewed as a special case in which each request forms its own stage. Second, stage-level optimization directly captures the straggler effect: completing a request earlier than others within the same stage does not reduce stage latency, yet may incur higher cost. Because the slowest request in each stage determines the stage completion time, the sequence of per-stage stragglers forms a natural proxy for the critical path of the entire DAG. Consequently, optimizing each stage's straggler approximates optimizing the global critical path of the DAG. Third, stage-level orchestration enables incremental refinement of global latency–cost trade-offs: after completing stage $k-1$, we accumulate the realized latency $L_{1:k-1}$ and cost $C_{1:k-1}$, and incorporate them into the optimization objective for the next stage, minimizing $OPT(\mathcal{G})_{1:k} = Latency(\mathcal{G})_{1:k} + \alpha \cdot Cost(\mathcal{G})_{1:k}$.

Specifically, within each execution stage, we determine the API provider assignment for each request by solving a stage-level integer linear program (ILP). Let $\mathcal{I}_k$ denote the set of requests in stage S_k, and let $\mathcal{P}$ denote the set of available API providers that serve the user-specified model.[2] For request $i \in \mathcal{I}_k$ and provider $p \in \mathcal{P}$, let L_i^p and C_i^p denote the predicted latency and monetary cost, respectively, as defined in §3.1. We introduce a binary decision variable $x_{i,p}$ indicating whether request i is assigned to provider p. The stage-level optimization problem is formulated as:

$$\min_{x, L_k} \quad \big(L_k + L_{1:k-1}\big) + \alpha \cdot \big(C_k + C_{1:k-1}\big) \quad (2)$$

$$\text{s.t.} \quad L_k \geq \sum_{p \in \mathcal{P}} L_i^p \, x_{i,p}, \quad \forall i \in \mathcal{I}_k, \quad (3)$$

$$\sum_{p \in \mathcal{P}} x_{i,p} = 1, \ \forall i \in \mathcal{I}_k, \ x_{i,p} \in \{0, 1\}. \quad (4)$$

where $C_k = \sum_{i \in \mathcal{I}_k} \sum_{p \in \mathcal{P}} C_i^p \, x_{i,p}$ denotes the total monetary cost of stage k, and Constraint (3) reports that the stage completion time is determined by the slowest request within the stage. As a result, SkyAPI incrementally compensates for earlier trade-off decisions (e.g., accumulated latency $L_{1:k-1}$ and cost $C_{1:k-1}$) when optimizing subsequent stages, thereby improving the global latency–cost trade-off. For example, if an earlier stage over-consumes cost due to output-length prediction errors, stage k can compensate by favoring lower-cost providers at the expense of modestly higher latency. Our extensive empirical evaluation shows that this stage-level optimization achieves performance close to an oracle with full graph information, while remaining robust to output-length prediction errors (§4.2).

[2]The model is fixed; only the provider varies.

3.2 Multi-Request Orchestration

Practical API providers often offer context caching mechanisms to reduce both latency and monetary cost. For example, the official DeepSeek API provides up to a 90% cost discount when requests hit the prefix cache (dee). In compound LLM serving, such prefix reuse is common both within a single compound request (e.g., inference-time scaling that explores multiple reasoning trajectories for the same input) and across different compound requests (e.g., shared system prompts).

Handling Shared Context. We next extend the formulation in Eq. (2) to account for context reuse across requests. First, we introduce a *prefix-warmup scheduling* mechanism. Specifically, among a group of parallel requests that process the same prompt prefix, one request is designated as the *leader* and issued immediately, initiating prefix generation and populating the provider's cache. The remaining requests defer submission by the TTFT of the leader, allowing the leader's prefix to be cached before they begin execution. This coordinated submission enables subsequent requests to hit the prefix cache and reuse generated prefix tokens without increasing their effective latency, since they would incur a comparable TTFT even without the warmup.

Second, following existing prefix-sharing mechanisms (e.g., SGLang (Zheng et al., 2024)), we maintain a lightweight trie that tracks prefix tokens across requests. When a prefix cache hit is detected,

we update request r's estimated latency and cost:

$$\hat{L}_r^p = \left(l_r^{in} - l_r^{cache}\right) \cdot T_p^{in} + l_r^{out} \cdot T_p^{out}, \quad (5)$$

$$\hat{C}_{m,p} = C_p^{cache} \cdot l_r^{cache} + C_p^{in} \cdot (l_r^{in} - l_r^{cache}) + C_p^{out} \cdot l_r^{out}. \quad (6)$$

where $C_{m,p}^{cache}$ denotes the costs of the cached input token. With these context-aware estimates, provider assignment proceeds unchanged using the stage-level optimization in Eq. (2).

Handling Latency Changes. Our extensive real-world measurements (§2.2) show that API processing speeds are typically stable over short time scales (seconds), although they may vary at longer time scales (minutes). This observation allows us to treat the per-token prefilling and decoding speeds, T_p^{in} and T_p^{out}, as stable within the execution of a request. In fact, even under transient latency changes, our stage-level optimization naturally compensates for them by incrementally correcting earlier decisions to preserve global cost effectiveness.

To further adapt to latency changes, SkyAPI maintains a running history of observed latencies for each provider. Upon the completion of a request, the system updates the provider's estimated processing speeds, T_p^{in} and T_p^{out}, which are then used to guide subsequent routing decisions.

Our empirical evaluation demonstrates that this design is robust to runtime uncertainty, achieving close-to-oracle performance (§4.2).

4 Evaluation

4.1 Experiment Setup

APIs, Models, and Workloads. We evaluate SkyAPI across ten real-world API providers: Parasail, Hyperbolic, Nebius, CentML, Microsoft Azure, Novita, GMI, SambaNova, Together.ai, and Kluster.ai. They collectively exhibit realistic diversity in API latency, throughput, and pricing structures. We use their APIs to access and evaluate across DeepSeek-R1, QwQ-32B, and Llama-3.3-70B models.

We evaluate SkyAPI on representative multi-agent system (MAS) benchmarks. Specifically, we use DeepResearch (Du et al., 2025), which focuses on information retrieval and multi-step reasoning, and GAMA-Bench (Huang et al., 2025), which evaluates MAS gaming workloads. Both benchmarks involve dynamic tool usage and planning. Because the performance of commercial APIs can vary over time (§2.2), we additionally develop a trace-driven simulator that records each provider's processing speeds over a 24-hour period. This simulator enables faithful and reproducible evaluation across all baselines under identical performance conditions. We will open-source the collected traces upon publication.

For both SkyAPI and SkyLLM (Zhao and Zhu, 2025), we fine-tune a BERT-based SLM to predict response lengths. We collect response-length statistics from the evaluated LLMs using a held-out set of requests for training, and report all final results on a disjoint test set of requests.

Baselines. We compare SkyAPI against industry-standard routing strategies and state-of-the-art academic advances, including three OpenRouter routing modes (ope, a):

- *OpenRouter-C*: A cost-minimization strategy that routes requests to the provider offering the lowest pricing, without considering latency.
- *OpenRouter-T*: A latency-optimized strategy that prioritizes providers with the highest decoding throughput.
- *OpenRouter-B*: The default OpenRouter strategy, which performs load-balanced routing across providers (e.g., round-robin) to balance cost and availability.
- *SkyLLM* (Zhao and Zhu, 2025): A state-of-the-art routing approach that selects providers for requests under a budget constraint while optimizing request latency, but does not account for inter-request dependencies.

Metrics. We evaluate the effectiveness of SkyAPI in terms of request completion time (i.e., end-to-end serving latency) and monetary cost, measured by the objective $\min\ \text{Latency}(\mathcal{G}) + \alpha \cdot \text{Cost}(\mathcal{G})$. We conduct ablation studies under different trade-off preferences (e.g., $\alpha \in \{1, 10, 1000\}$) in Section 4.3. For each experiment, we report the mean value over 10 independent runs.

4.2 End-to-End Performance

SkyAPI improves the cost–latency frontier. Figure 6 compares total monetary cost as the trade-off parameter α varies from 10^1 to 10^4 in the minimizing $OPT(\mathcal{G}) = \text{Latency}(\mathcal{G}) + \alpha \cdot \text{Cost}(\mathcal{G})$. Across the entire spectrum, SkyAPI consistently achieves the lowest cost, demonstrating a strictly improved cost–latency frontier over all baselines.

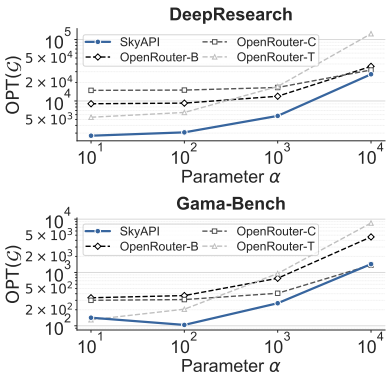

Figure 6: SkyAPI improves cost-latency frontier across different trade-off preferences (α). (DeepSeek-R1)

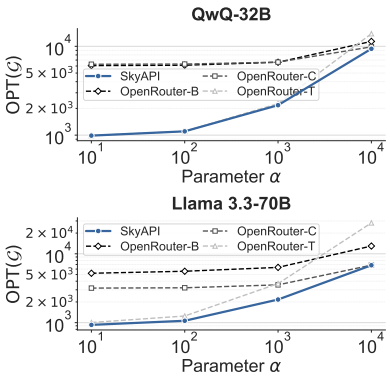

Figure 7: SkyAPI improves cost-effectiveness across models and tradeoff preferences (DeepResearch).

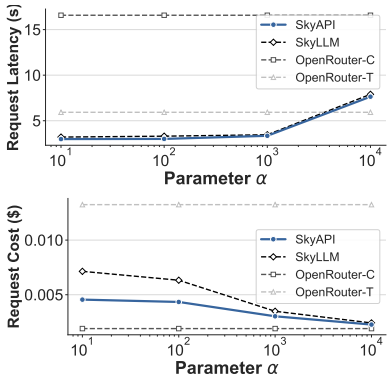

Figure 8: SkyAPI optimizes the balance between cost and latency (DeepSeek-R1 and DeepResearch).

Figure 8 dissects the trade-off by plotting request latency and monetary cost separately against α. In the low-α regime, SkyAPI demonstrates significant efficiency gains over the state-of-the-art baseline, SkyLLM. Specifically, SkyAPI reduces monetary cost by approximately 37% compared to SkyLLM while maintaining slightly lower end-to-end latency. As α increases, SkyAPI consistently maintains a superior Pareto frontier. This advantage stems from our stage-level orchestration, which identifies non-critical execution paths and routes them to cost-effective providers without compromising the critical path.

SkyAPI delivers consistent improvements across tasks and models. We next extend our evaluation to Gama-Bench (Huang et al., 2025). The results, shown in Figure 6 (bottom), confirm that SkyAPI maintains the performance advantage. In the low-α regime, SkyAPI reduces the cost by about 50% compared to the OpenRouter-T baseline. This suggests that our stage-level optimization identifies non-obvious, low-cost routing paths that cost-only minimizers overlook.

We further explore the performance of SkyAPI across DeepSeek-R1, *QwQ-32B*, and *Llama-3.3-70B* models using the DeepResearch benchmark. Figure 7 reports that SkyAPI delivers consistent and significant improvements across models under a wide spectrum of the trade-off parameter α. In the low-α regime, SkyAPI significantly undercuts the OpenRouter-C and OpenRouter-B baselines, reducing expenses by approximately $6\times$ to $3\times$ for both QwQ and Llama-3.3.

SkyAPI achieves close-to-oracle performance. To assess the performance ceiling, we compare SkyAPI against an oracle setting equipped with perfect output-length information (i.e., zero prediction error). Figure 10 reveals a marginal optimality gap of less than 5%, confirming that SkyAPI remains robust to prediction errors and approaches the theoretical cost–latency limit.

SkyAPI introduces negligible overheads. To ensure that our orchestration does not become a bottleneck, we profiled the computational latency of SkyAPI's core components: the BERT-based length predictor and the ILP solver. Table 1 details the average per-request overhead across benchmarks. The cumulative overhead accounts for less than 2% of the total end-to-end duration. This confirms that SkyAPI is highly lightweight and can be deployed in real-time serving systems with negligible impact on system latency.

Table 1: Average per-request runtime overhead of SkyAPI components (in milliseconds).

Benchmark	Predictor	ILP Solver
DeepResearch	45.7	10.7
Gama-Bench	32.8	14.4

4.3 Ablation Studies

Breakdown of Request Performance. To isolate the contribution of output-length prediction, we conduct an ablation study in which the predictor and the scheduler are separately disabled. In the first variant, the output-length predictor is ablated by replacing the estimated length $\hat{l}_{\text{out}}$ with a random value sampled from the empirical distribution of historical response lengths. In the second variant, we treat requests independently without

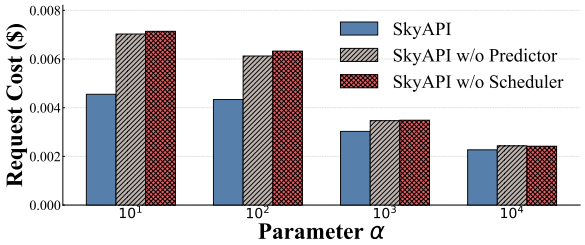

Figure 9: Performance breakdown of SkyAPI.

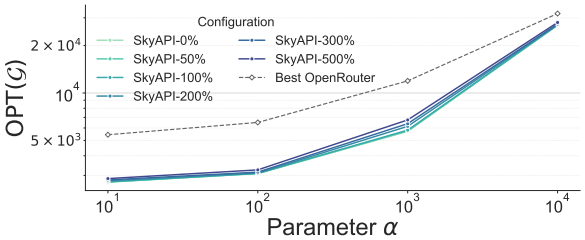

Figure 10: SkyAPI achieves consistent improvements under different degrees of length prediction errors.

considering their interdependency.

Figure 9 shows that disabling the length predictor and the scheduler increases total cost by approximately 10% across all values of α. Notably, the performance degradation remains modest, highlighting the resilience of our stage-level optimization: by incrementally compensating for prediction errors across stages, SkyAPI preserves near-optimal global cost effectiveness.

Impact of Prediction Errors. On the DeepResearch benchmark, our output-length prediction model achieves a mean absolute error (MAE) of 120 tokens and a root mean squared error (RMSE) of 191 tokens. Given that reasoning tasks often generate thousands of tokens, these results indicate that the predictor is effective.

To further assess the practical robustness of SkyAPI, we conduct ablation studies under controlled output-length prediction errors: Error Rate $= \frac{|\hat{l}-l|}{l} \times 100\%$, where $\hat{l}$ is the predicted output length and l is the ground-truth length. In Figure 10, *Best OpenRouter* denotes the best-performing configuration among the three OpenRouter baselines for each value of α. Across all simulated error levels, SkyAPI consistently outperforms all baselines. Notably, practical SkyAPI (e.g., with 50% prediction error) already approaches oracle-level performance with 0% prediction error, further validating the effectiveness of our incremental stage-level optimization design.

5 Related Work

Agentic Frameworks. Recent systems such as LangChain (LangChain, 2025) enable sequential task execution by chaining LLM calls with external tools and data sources. More structured approaches, including AgentOrchestra (Wang et al., 2025b) and AgentFlow (Li et al., 2025), employs a high-level planner that coordinates specialized sub-agents through dynamically constructed execution DAGs. These frameworks focus on expressiveness and modularity of agent behaviors. In contrast, SkyAPI complements such frameworks by enabling *cost-effective and adaptive request orchestration* across dependent agent invocations.

LLM Routing. LLM-Blender (Jiang et al., 2023) focuses on ensemble routing, ranking outputs from multiple models to select the best response. RouteLLM (Ong et al., 2024) and MixLLM (Wang et al., 2025a) train learned routers to select an appropriate model for each query based on its characteristics. SkyLLM (Zhao and Zhu, 2025) optimizes accuracy, latency, and cost at the per-request level. Instead, SkyAPI explicitly targets *dependent and structured request graphs* arising in MAS, and addresses additional practical challenges such as context sharing and runtime dynamics.

Prediction for LLM Serving. Early work (Zheng et al., 2023) demonstrated that LLMs themselves can estimate response length. Subsequent approaches (Shahout et al., 2024; Zhang et al., 2026) use learned predictors to inform scheduling and batching decisions. These systems primarily optimize isolated requests under controlled environments. In contrast, SkyAPI addresses prediction and scheduling in the presence of *inter-request dependencies and workflow-level constraints*.

6 Conclusion

This paper presented SkyAPI, a structure-aware API orchestration framework for compound LLM serving. By modeling agentic workflows as execution graphs and optimizing provider selection at the stage level, SkyAPI overcomes the limitations of per-request routing under inter-request dependencies, execution-time uncertainty, and provider heterogeneity. Our design integrates incremental optimization, prefix-warmup scheduling, and robustness to prediction errors, achieving 54% cost savings while reducing latency by 62%.

7 Limitations

First, due to resource constraints, our evaluation is not able to include large-scale, fully live API invocations. As a result, it may not capture rare but transient network anomalies—such as packet loss, sudden connection timeouts, or short-term jitter—that can arise in public Internet environments. Consequently, the reported end-to-end latency may differ slightly from that observed in production-scale deployments.

Second, SkyAPI focuses on optimizing API selection for a fixed model, thereby preserving response quality by design. While this avoids quality regression, it limits opportunities for semantic-aware routing that explicitly trades off model quality against efficiency. Incorporating such quality-aware selection would require accurately predicting and validating response quality at runtime, which introduces nontrivial overhead. We leave the integration of lightweight quality-aware routing signals as an important direction for future work.

References

Deepseek: https://api-docs.deepseek.com/quick_start/pricing. https://api-docs.deepseek.com/quick_start/pricing.

a. Openrouter: Auto router. https://openrouter.ai/docs/guides/features/routers/auto-router.

b. Openrouter: The unified interface for llms. https://openrouter.ai/.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. 2025. Deepresearch bench: A comprehensive benchmark for deep research agents. *arXiv preprint arXiv:2506.11763*.

Mingyan Gao, Yanzi Li, Banruo Liu, Yifan Yu, Phillip Wang, Ching-Yu Lin, and Fan Lai. 2025. Single-agent or multi-agent systems? why not both? *Preprint*, arXiv:2505.18286.

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

Jen-tse Huang, Eric John Li, Man Ho Lam, Tian Liang, Wenxuan Wang, Youliang Yuan, Wenxiang Jiao, Xing Wang, Zhaopeng Tu, and Michael Lyu. 2025. Competing large language models in multi-agent gaming environments. In *The Thirteenth International Conference on Learning Representations*.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. *arXiv preprint arXiv:2306.02561*.

LangChain. 2025. Langchain. Python library for building LLM-powered applications. Accessed 2025-10-15.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Zou, and Pan Lu. 2025. In-the-flow agentic system optimization for effective planning and tool use. *arXiv preprint arXiv:2510.05592*.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. 2024. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186.

Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. 2024. Don't stop me now: Embedding based scheduling for llms. *arXiv preprint arXiv:2410.01035*.

Xinyuan Wang, Yanchi Liu, Wei Cheng, Xujiang Zhao, Zhengzhang Chen, Wenchao Yu, Yanjie Fu, and Haifeng Chen. 2025a. Mixllm: Dynamic routing in mixed large language models. *arXiv preprint arXiv:2502.18482*.

Yiheng Wang, Weijia Chen, Wenhao Yang, Ke Xu, and 1 others. 2025b. Agentorchestra: Hierarchical multi-agent orchestration with tool management for gaia benchmark. https://arxiv.org/abs/2506.12508. ArXiv:2506.12508, accessed 2025-10-15.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, and 1 others. 2024. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*.

Wei Zhang, Zhiyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. 2026. Jitserve: Slo-aware llm serving with imprecise request information. *NSDI*.

Heng Zhao and Yifei Zhu. 2025. Skyllm: Cross-llm-apis federation for cost-effective query processing. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 20864–20873.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. Sglang: efficient execution of structured language model programs. NeurIPS.

Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. 2023. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems*, 36:65517–65530.