RouteWise: Latency–Cost Optimization for Multi-Provider LLM Routing

Muxin Tian*† University of Toronto
Haoran Ni* Harvard University
Yiyan Zhai† Carnegie Mellon University
Yangsun Park† Tufts University
Juncheng Yang Harvard University

Abstract

Open-weight large language models (LLMs) are increasingly served by multiple providers, creating an opportunity to route requests across providers to reduce inference cost and latency. Exploiting this market is challenging for two reasons. First, providers use heterogeneous pricing schemes including on-demand APIs, quota plans, and concurrency subscriptions that are not directly comparable. Second, provider latency varies substantially over time, requiring routing decisions to balance cost against performance. Existing routers either trade model quality for cost by routing across different models, or optimize cost or latency in isolation without accounting for subscription capacity.

We present RouteWise, a multi-provider routing system that jointly optimizes cost and latency for a fixed LLM. RouteWise develops a unified cost model that converts on-demand spending and scarce subscription capacity into a common effective monetary cost, allowing prepaid capacity to be allocated to requests where it yields the greatest savings. On top of this model, a provider mixer solves a linear program to minimize mean time-to-first-token (TTFT) under a configurable cost budget, while probability-targeted hedging reduces tail-latency SLO violations without eroding cost savings. A single operator-tunable knob controls the cost–latency trade-off. We implement RouteWise as a practical LLM request router and evaluate it on BurstGPT and a production agentic workload from FreeInference. Compared with OpenRouter's automatic routing, RouteWise's cost-oriented operating point reduces serving cost by 18.9%, mean TTFT by 23.2%, and SLO violations by 74.6%.

1 Introduction

LLM inference cost has become a major barrier to AI adoption. For startups, API bills can rival engineering costs; for universities, limited research credits can restrict student projects, coursework, and experimentation [27]. A common response is to switch to cheaper, smaller models, but this often sacrifices the quality that makes LLMs useful in the first place [18, 22].

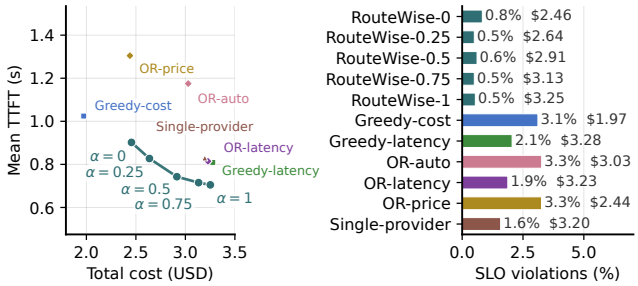


Figure 1. Cost–latency trade-off on a 24-hour BurstGPT replay of MiniMax-M3. (Left) Total serving cost versus TTFT; lower left is better. Routewise provides a control knob to balance between cost and latency. (Right) Fraction of requests exceeding the 3-second TTFT target; labels give total cost. Greedy-cost and Greedy-latency optimize one objective; OR-auto, OR-price, and OR-latency are OpenRouter routing modes [33], and Single-provider sends every request to the best on-demand provider.

At the same time, the LLM serving market has moved beyond simple on-demand APIs. The commoditization of open-weight models such as Llama-3.3-70B [25], MiniMax-M2.5 [31], and DeepSeek-V4 [9] means that the *same model* is now available from many providers under substantially different pricing schemes. On-demand APIs and marketplaces, such as Together [41] and OpenRouter [33], provide elastic access but charge per token. Subscription services instead offer bounded capacity at zero monetary marginal cost, either through request quotas, such as Chutes.ai's 5,000 requests/day plan [7], or through reserved concurrency, such as Featherless.ai's C concurrent slots [12].[1] This pricing diversity creates an opportunity: *routing requests across providers that serve identical models can reduce cost without changing model quality.*

Realizing this opportunity is non-trivial. A naive greedy policy that exhausts subscription capacity before using metered APIs ignores the *effective cost* of scarce subscription capacity. Using a quota slot for a short, inexpensive request may force a later, more expensive request onto a metered provider. On our 30-day replay of 1.81 million BurstGPT

*Equal Contribution.
†Work done at Harvard University.

[1]While concurrency-based subscription is less popular than quota-based subscription, local-deployment can be modeled as a concurrency-based provider.

arrivals paired with ShareGPT requests, using a quota subscription plus metered fallback, this greedy policy costs 19.3% more than the offline optimum, a gap RouteWise narrows to 15.0% (§ 4.4.1).

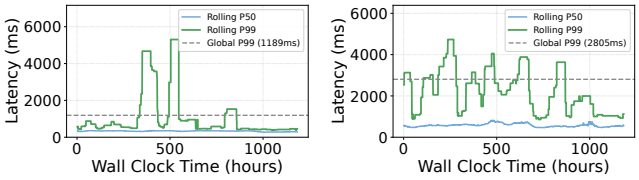
Cost is not the only concern. Subscription providers can have higher or more variable latency, and provider latency is often non-stationary. We measured the time-to-first-token (TTFT) of Llama-3.3-70B using Meta's Llama API and GPT-4o-mini using OpenAI API over 50 days (Figure 2) and found several-fold variation in rolling P99 latency. Consequently, blindly prioritizing low-cost subscriptions can substantially degrade user experience. Cost-only routing is therefore insufficient for interactive applications with strict service-level objectives (SLOs) [8].

Although routing is common in LLM serving, existing routers are not designed for this setting. Model routers and cascade systems typically reduce cost by selecting among *different* models, thereby trading off cost and response quality [6, 30]. Marketplaces such as OpenRouter expose multiple providers for the same model and support provider sorting and fallbacks [32, 35]. However, their routing policies focus exclusively on a single objective at a time—such as price, latency, or throughput—and fail to provide guarantees for any other objective. In contrast, we study provider routing for a fixed model: the model quality is held constant, and we propose a routing algorithm that jointly optimizes cost and latency performance.

We designed RouteWise, a routing system that jointly optimizes cost and latency for multi-provider LLM routing. RouteWise has two parts. First, a *cost layer* that computes an effective per-request price for each provider, normalizing across heterogeneous pricing schemes. Then, a *latency layer* that selects a provider to minimize mean TTFT within a cost budget users set to balance latency against cost. To reduce tail latency and SLO violation, RouteWise introduces a smart hedging that only sends a backup request if the primary request may violate SLO while the backup can meet SLO.

We implemented RouteWise in an LLM inference router and a simulator, and evaluated it on two real-world workloads: BurstGPT and FreeInference agentic workloads [13]. Our main contributions are:

- We introduce a unified cost model for LLM routing across on-demand, quota, and concurrency pricing. Our model converts scarce subscription capacity into an effective monetary cost using online threshold pricing, enabling a single routing objective that allocates prepaid capacity to the requests where it delivers the greatest savings.
- We develop a routing algorithm that jointly optimizes cost and latency. A provider mixer solves a linear program (LP) to minimize mean TTFT under a cost budget controlled by a single operator-tunable knob, while probability-targeted smart hedging reduces tail SLO violations without sacrificing the resulting cost savings.



(a) Meta: Llama-3.3-70B **(b)** OpenAI: gpt-4o-mini

Figure 2. Provider latency over 50 days (1,200 hours) for fixed 10-token prompts. We issue two probes per hour and compute each rolling P50 and P99 from the trailing 100 observations (nominally 50 hours); the dashed line is the P99 over the full trace. Rolling P99 fluctuates substantially: Meta's Llama-3.3-70B ranges from 1.2–5.4s, while OpenAI's GPT-4o-mini ranges from 1.0–4.8s. Each rolling estimate therefore contains 100 samples rather than the 30 samples in the router's separate 15-minute online profile.

Table 1. Different LLM serving pricing models. Light-use cost measures the monthly cost of 1 request of 1k input and 200 output tokens every day. Heavy-use cost measures the monthly cost for 5k requests with 120k input tokens and 8k output tokens. On-demand price assumes the MiniMax-M2.5 model.

Type	Sym.	Example	Light	Heavy
On-demand	$\mathcal{P}_O$	OpenRouter [31]	$0.01	$138
Quota	$\mathcal{P}_Q$	Chutes.ai [7]	$20	$20
Concurrency	$\mathcal{P}_C$	Featherless.ai [12]	$25	$25

- We implement RouteWise as a Python library for LLM request routing[2] and evaluate it on BurstGPT and a production agentic request trace from FreeInference. Compared with OpenRouter's automatic routing, RouteWise's cost-oriented configuration reduces cost by 18.9%, mean TTFT by 23.2%, and SLO violations by 74.6%.

2 Background

LLM inference serves user requests by running a pre-trained model to transform an input prompt into generated output tokens. Unlike traditional stateless web services, an LLM inference request has two distinct computational phases. In the *prefill* phase, the system processes the full input prompt and builds the model's key-value cache. In the *decode* phase, the system generates output tokens autoregressively, one token at a time. The cost and latency of a request therefore depend not only on the number of requests, but also on the input length, output length, model size, batching policy, cache reuse, and provider load.

2.1 Diverse LLM Inference Pricing Models

The commoditization of open-weight LLMs, such as MiniMax-M2.5 [31], Kimi-K2.6 [28], and DeepSeek-V4 [9], means that the *same* model is now available from many providers under different pricing structures. We identify three common

[2]The RouteWise library is available at https://github.com/HarvardMadSys/RouteWise. It is also integrated into HybridInference (https://github.com/HarvardMadSys/hybridInference), the open-source LLM gateway that powers FreeInference.

pricing models that cover the current provider marketplace: on-demand billing, quota subscriptions, and concurrency subscriptions (Table 1).

On-demand billing. On-demand providers offer elastic access, charging per input and output token without a hard capacity limit. For example, OpenRouter prices MiniMax-M2.5 at $0.15 per million input tokens and $1.20 per million output tokens [31]. Cost therefore scales with request length: a short "hello" query costs a fraction of a cent, while a long coding or reasoning task can cost orders of magnitude more. Many providers also charge discounted rates for cached input tokens to reflect prompt-prefix reuse [34].

Quota subscriptions. Quota-based subscriptions provide a fixed amount of usage per reset window for a flat recurring fee. The quota may be expressed in requests, tokens, or other provider-specific units. For instance, Chutes.ai offers 5,000 requests per day at $20/month [7]; Cerebras Code offers 24 million tokens per day at $50/month [5]; and Ollama Cloud meters usage in cloud GPU time, offering $20/month and $100/month tiers with 5-hour sessions and weekly reset windows [29]. Once the quota is exhausted, subsequent requests must either wait until the next reset window or pay per token.

Concurrency subscriptions. Concurrency-based subscriptions reserve serving capacity for a fixed number of simultaneous requests. For example, Featherless.ai offers 4 concurrent connections at $25/month [12], and Arli AI offers 2, 6, 20, and 40 parallel-request tiers at $25, $75, $250, and $1,000/month, respectively [3]. Unlike quota subscriptions, these plans bound *simultaneous usage* rather than total request count: each request occupies a slot for its full duration, and the slot cannot serve another request until it completes.

2.2 LLM Inference Latency

LLM serving latency is commonly decomposed into time to first token (TTFT) and time per output token (TPOT). These metrics capture different parts of the inference pipeline and matter for different user experiences. TTFT determines how quickly the user sees the first response, while TPOT determines how quickly the rest of the response streams after generation begins. In interactive LLM applications, TTFT is often the more visible latency component because users perceive the system as responsive once streaming starts.

TTFT and TPOT. TTFT measures the elapsed time from when a request is submitted until the first output token is returned. It includes request routing, queueing, batching delay, input prefill, and generation of the first token. TPOT measures the average time between subsequent output tokens during the decode phase. A request's end-to-end latency can be viewed as the sum of TTFT and the cumulative decode time for the remaining output tokens. Thus, TTFT captures startup latency, while TPOT captures streaming throughput.

Mean TTFT is important. Mean TTFT is a critical metric for interactive LLM applications. In chat workloads, users typically read streamed text more slowly than models generate it, so improving decode throughput beyond the user's reading rate may provide limited perceptual benefit. By contrast, reducing TTFT directly improves perceived responsiveness because it shortens the initial blank wait before any output appears. TTFT is also important for agentic workloads, where requests often involve short outputs but repeated model calls. In these settings, end-to-end task latency is heavily affected by the startup delay of many sequential invocations rather than only by long generations. In our production agentic traces, Figure 3a shows median TTFT shares of 54.6%, 73.1%, and 97.1% across GPT-5.4, Claude Opus 4.7, and MiniMax-M2.5, respectively, making mean TTFT a first-order performance objective.

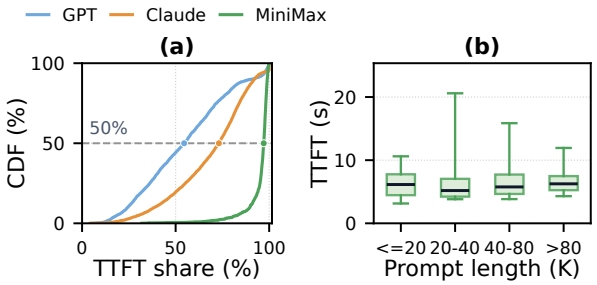


Figure 3. Production traces show why TTFT is central to agentic routing. (a) Empirical CDF of TTFT as a share of end-to-end request duration; dots mark provider means, while each curve's intersection with the dashed 50% line gives its median. (b) Production TTFT by prompt length, with boxes showing the 25th–75th percentiles and whiskers showing the 5th–95th percentiles. Tail TTFT is not monotonic in prompt length, consistent with provider-side queueing outliers.

TTFT tail is dominated by queueing rather than context length. Tail TTFT is especially important because long startup delays can violate service-level objectives even when average latency is acceptable. Unlike decode latency, which scales strongly with output length, TTFT is primarily affected by routing delay, provider load, queueing, batching, and prefill cost. In production, Figure 3b shows that the TTFT tail does not simply grow with prompt length, indicating that queueing-induced provider outliers are a major source of tail startup delay. This makes TTFT a useful target for SLOs: it captures provider responsiveness and queueing behavior without being dominated by the total length of the generated response. Consequently, RouteWise optimizes both mean TTFT and tail TTFT, using provider selection to reduce average startup delay and smart hedging to mitigate severe queueing-induced outliers.

2.3 Motivation: Routing for Low Cost and Latency

Subscriptions benefit both sides of the market. Users get predictable spending and lower marginal cost for steady workloads, while providers get recurring revenue and a clearer signal for capacity planning. Quota and concurrency limits

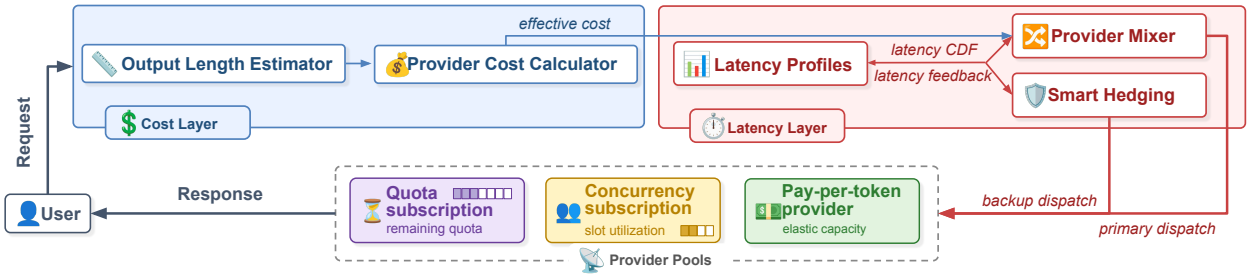


Figure 4. RouteWise system architecture. The cost layer assigns each provider a unified effective cost that evaluates both monetary spend and scarce subscription capacity. The latency layer selects the min-latency provider mix subject to a per-request cost budget $B_\alpha(t)$, and smart hedging dispatches a backup at the latest time such that the combined primary-plus-backup SLO-success probability still meets a target.

make these plans operationally tractable by bounding total usage or simultaneous load.

This pricing diversity creates a routing opportunity. When the *same* model is available through on-demand APIs, quota subscriptions, and concurrency subscriptions, the cheapest long-term choice is not necessarily the cheapest provider for the current request. A router must decide when to spend prepaid quota, when to occupy reserved concurrency, when to use elastic on-demand service, and when to preserve prefix-cache locality for future turns.

Latency complicates the decision further. Providers serving the same model can have different and rapidly changing TTFT distributions, often due to provider-side queueing. A cheapest-first policy may violate interactive latency SLOs, while always choosing the fastest provider or aggressively hedging requests can eliminate the cost savings from subscriptions and cached-token discounts.

Multi-provider LLM routing is therefore a joint online cost–latency control problem. § 3.2.1 formalizes this objective. RouteWise decomposes the problem into a cost layer that computes effective request prices across heterogeneous pricing and cache states (§ 3.2), and a latency layer that selects and hedges providers under TTFT SLOs (§ 3.3).

3 RouteWise Design

This section presents RouteWise, a cost- and latency-aware routing system for multi-provider LLM inference.

3.1 System Overview

Figure 4 shows the RouteWise architecture. An incoming request passes through two layers that jointly select a provider. The **cost layer** assigns each provider an effective cost, translating both monetary token pricing and bounded subscription capacity onto a single unified scale. The **latency layer** then picks the lowest-latency provider mix whose effective cost stays within an operator-set budget. Once the request is in flight, smart hedging protects the SLO target by dispatching a backup when the primary is unlikely to meet the deadline.

Cost Layer (§ 3.2). The cost layer makes heterogeneous pricing models directly comparable. Its output length estimator and provider cost calculator map on-demand prices, depleting quotas, and idle concurrency slots onto a single monetary scale, so that scarce subscription capacity is spent on the requests where it saves the most.

Latency Layer (§ 3.3). The latency layer turns effective costs into routing decisions that meet latency SLOs. Its provider mixer minimizes average TTFT within an operator-set cost budget $B_\alpha(t)$, while smart hedging protects the latency tail by dispatching a backup only when a request risks violating its SLO.

Request Flow. When a request arrives, RouteWise processes it in four steps:

1. **Length Estimation:** The output length estimator predicts the request's generated token count to gauge its baseline on-demand API value.
2. **Cost Unification:** The provider cost calculator evaluates current provider utilization and translates disparate billing models into a single normalized effective cost for each candidate.
3. **Latency Routing:** The provider mixer solves a lightweight LP, once per arriving request, using empirical latency profiles and effective costs to decide a cost-budgeted, min-latency provider. Any consumed quota or concurrency slot is immediately tracked upon dispatch.
4. **Smart Hedging:** While the request is in flight, smart hedging periodically checks the elapsed time against the target SLO. At the last checkpoint that can still meet the target success probability, it dispatches the cheapest feasible backup and returns the first response to the user.

3.2 Cost Layer

3.2.1 Problem Statement. Requests arrive online in a continuous stream. Each request is characterized by a known input length and an unknown output length, which together determine its final token cost. Every incoming request needs to be assigned to one provider from three available categories:

- **On-demand providers**: incur a direct *monetary marginal cost* proportional to the total number of input and output tokens processed. There are no capacity limits.
- **Quota subscriptions**: incur zero monetary marginal cost per request, but the provider allows at most a fixed number of requests to be served within each billing window.
- **Concurrency subscriptions**: incur zero monetary marginal cost per request, but the provider restricts the maximum number of requests that can be processed simultaneously in flight.

The system's objective is to minimize the total incurred on-demand API cost over time, treating the upfront subscription fees as sunk costs. To do this elegantly across all three categories, RouteWise maps every provider to a unified **effective cost**. For on-demand providers, the effective cost is exactly the expected monetary marginal cost; for subscription providers, it reflects the dynamic valuation of consuming bounded capacity.

3.2.2 Offline Analysis. To guide our online design, we first analyze the offline setting where the full request sequence is known *a priori*. This allows us to establish theoretical cost lower bounds and expose the underlying structure of optimal routing. We first isolate quota limits and then add concurrency limits.

Quota and On-Demand. Suppose the only scarce subscription resource is a provider with a per-window quota Q. Because each request consumes exactly one quota slot at zero marginal cost, the optimal strategy simply routes the Q requests that would have incurred the highest on-demand API costs to the quota provider, sending the remainder to the metered API.

Theorem 3.1 (Greedy-by-Value Optimality)**.** *For quota-only routing with unit request weights, routing the Q requests with the highest API costs to the quota subscription is optimal and computable in $O(n \log n)$ time.*

This structural insight—that optimal offline routing is strictly greedy by value—directly motivates our online approach. It implies that an online algorithm should only admit a request to a subscription slot if its expected value is sufficiently high, which we formalize using a dynamic effective cost threshold.

Adding Concurrency Limits. When we introduce a concurrency-based subscription, the problem becomes significantly more complex, coupling quota allocation with parallel machine scheduling.

Theorem 3.2 (NP-Hardness)**.** *Under deadline or bounded-waiting constraints, the combined quota-and-concurrency routing problem is NP-hard. This holds even in a simplified setting with a single concurrency slot (C=1), no quota (Q=0), and identical arrivals, via a reduction from the 0/1 knapsack problem.*

To compute offline optimums despite this hardness, we formulate a time-indexed Integer Linear Program (ILP) that scales to thousands of requests within minutes using Gurobi [15]. These ILP solutions serve as oracle baselines in our evaluation (§ 4). Furthermore, the difference between the quota-only and combined optimums exposes the *cost of concurrency*: the inevitable increase in API spend caused by physical resource contention, which forces high-value requests to spill over to on-demand providers when concurrency slots are saturated.

3.2.3 Online Routing. In the online setting, we must make irrevocable routing decisions without knowledge of future arrivals or exact output lengths. We tackle this by introducing a unified *effective cost* that standardizes disparate billing methods—on-demand pricing, quota limits, and concurrency constraints—onto a single monetary scale, dynamically adjusting subscription effective costs as capacity is consumed.

Quota Effective Cost. For quota provider j, let Q_j be its per-window request quota and $u_j \in [0, 1]$ the fraction of that quota consumed in the current window. Usage resets at each window boundary. We define an exponential threshold:

$$L \cdot \left(\frac{U}{L}\right)^{u_j} \tag{1}$$

where L and U represent a workload's cost range. For each request, we compute the lowest cost of serving it through any on-demand provider, representing the cost avoided by using subscription capacity instead. Across the entire workload, L and U are then set to the 10th and 90th percentiles of these per-request costs. In deployment the workload is not known in advance, so L and U can be estimated from recent traffic history. Intuitively, the threshold starts low (accepting most requests) and rises exponentially as quota is consumed, reserving remaining slots for increasingly valuable requests. This exponential increase is bounded: because $u_j \in [0, 1]$, the threshold always lies in $[L, U]$.

Quota-only routing (§ 3.2.2) maps each quota provider to the online knapsack problem: each arriving request is an item with value $v_i \in [L, U]$ (the avoided API cost), the quota Q_j is the capacity, and admit/reject must be committed at arrival time. This exponential threshold is the standard primal-dual solution for this problem, with the following guarantee:

Theorem 3.3 (Competitive Ratio [49, Theorem 2.1])**.** *The primal-dual algorithm with exponential threshold achieves a competitive ratio of $\ln(U/L) + 1$ for the online knapsack problem with unit weights and known item values in $[L, U]$.*

Concurrency Effective Cost. Unlike depletable quotas, concurrency slots are *reusable*: they instantly regenerate upon request completion but cannot be banked. Because idle slots are wasted, we assign an available slot an effective cost of zero, ensuring it is maximally utilized before incurring any on-demand API costs. Saturation is handled purely as a hard

admission constraint: when all slots are occupied, provider j is temporarily removed from the feasible set $\mathcal{P}^{\mathrm{f}}(t)$, deflecting traffic elsewhere. Thus, concurrency requires only binary momentary rate-limiting rather than exponential rationing.

On-Demand Effective Cost. For on-demand APIs charging per token ($j \in \mathcal{P}_O$), the effective cost is the expected monetary price of the request, explicitly accounting for provider-specific prefix caching discounts. For an arriving request with input length n_{in} and predicted output length n_{out}, let n_j^{cache} be the estimated number of cached input tokens if routed to provider j. The on-demand effective cost evaluates to:

$$c_j^{\mathrm{OD}} = p_j^{\mathrm{in}}(n_{\mathrm{in}} - n_j^{\mathrm{cache}}) + p_j^{\mathrm{cache}} n_j^{\mathrm{cache}} + p_j^{\mathrm{out}} n_{\mathrm{out}} \tag{2}$$

where p_j^{in}, p_j^{out}, and p_j^{cache} are the provider's standard input, output, and discounted cached-input token prices. If a provider does not offer a cache discount, we set $p_j^{\mathrm{cache}} = p_j^{\mathrm{in}}$; if no cache-hit signal is available, we set $n_j^{\mathrm{cache}} = 0$. The cache term affects only effective cost; latency routing still relies on observed TTFT distributions.

Unified Effective Cost. The cost layer outputs a single per-provider effective cost scalar c_j that concisely unifies the three pricing models:

$$c_j = \begin{cases} c_j^{\mathrm{OD}} & j \in \mathcal{P}_O \\ L \cdot \left(\frac{U}{L}\right)^{u_j} & j \in \mathcal{P}_Q \\ 0 & j \in \mathcal{P}_C \end{cases} \tag{3}$$

The first case is the on-demand API price, the second evaluates the exponential quota effective cost at the current utilization u_j, and the third assigns a zero effective cost for any available concurrency slot. All three cases share the same monetary scale set by the API cost envelope $[L, U]$, allowing the latency layer to compare providers directly (§ 3.3).

3.2.4 Output Length Estimator.

Because generation length is unknown at routing time, RouteWise uses a coarse estimate of n_{out} to compute the expected on-demand API cost (Equation 2) and compare it against subscription effective costs. We use a lightweight online bucket-mean estimator: requests are grouped into log-spaced buckets by input length, and each request is assigned the running mean of its bucket's historical output lengths. If a bucket lacks samples, it falls back to the global running mean.

This simple, featureless point estimator is sufficient in our workloads because output is short relative to input and cost is dominated by input size (§ 4.4). Thus, we deliberately avoid the complexity of training high-fidelity LLM response-length predictors.

3.3 Latency Layer

The latency layer selects a provider across $\mathcal{P}_O$, $\mathcal{P}_Q$, and $\mathcal{P}_C$ using the effective cost c_j from Equation 3. Because provider latency is non-stationary (Figure 2), this decision must adapt to recent observations. Our measurements reveal an important distinction: typical latency (e.g., P50 or mean) is predictable over short time horizons, while tail latency (e.g., P99) decorrelates quickly and is difficult to predict. This motivates a two-part design. The LP uses the rolling mean TTFT $\bar{t}_j$ to proactively select a provider, while smart hedging (§ 3.3.3) handles unpredictable tail events reactively: as a request runs, it dispatches a backup when the estimated primary-plus-backup probability of meeting the SLO falls below the target.

3.3.1 Online Latency Profiling.

To make routing decisions, RouteWise needs an up-to-date estimate of each provider's latency before dispatching a request. For each provider, it maintains a 15-minute rolling window of recently observed TTFT measurements. The router uses the mean TTFT to favor lower-latency providers, while the hedging mechanism (§ 3.3.3) uses the empirical latency distribution to estimate whether an in-flight request is likely to meet its SLO. Observations come primarily from live traffic; when a provider is idle, lightweight background probes keep its profile from becoming stale. This allows RouteWise to continuously adapt its routing decisions as provider performance changes over time.

3.3.2 LP-Based Provider Mixer.

Let $\mathcal{P} = \mathcal{P}_O \cup \mathcal{P}_Q \cup \mathcal{P}_C$ be the full set of candidate providers, each with effective cost c_j from Equation 3. Let $\mathcal{P}^{\mathrm{f}}(t) \subseteq \mathcal{P}$ denote the feasible set of providers at decision time t (i.e., those not masked out by quota exhaustion or concurrency saturation). We solve for routing fractions π_j (the probability of dispatching the request to provider j) that minimize expected latency subject to a per-request cost budget:

$$\min_{\pi} \quad \sum_{j \in \mathcal{P}^{\mathrm{f}}(t)} \pi_j \, \bar{t}_j \tag{4}$$

$$\text{s.t.} \quad \sum_{j \in \mathcal{P}^{\mathrm{f}}(t)} \pi_j \, c_j \le B_\alpha(t) \tag{5}$$

$$\sum_{j \in \mathcal{P}^{\mathrm{f}}(t)} \pi_j = 1, \quad \pi_j \ge 0$$

where $\pi_j \in [0, 1]$ is the routing fraction, $\bar{t}_j$ is the rolling-window mean TTFT of provider j, and c_j is the per-provider effective cost from the cost layer. The cost budget linearly interpolates between the cheapest and most expensive feasible providers at the current decision time:

$$B_\alpha(t) \;=\; (1-\alpha) c_{\min}(t) \;+\; \alpha c_{\max}(t), \tag{6}$$

where $c_{\min}(t) = \min_{j \in \mathcal{P}^{\mathrm{f}}(t)} c_j$ and $c_{\max}(t) = \max_{j \in \mathcal{P}^{\mathrm{f}}(t)} c_j$. The parameter $\alpha \in [0, 1]$ is a normalized Pareto knob: $\alpha = 0$ tightens the budget to the cheapest feasible provider (minimum cost, no latency tradeoff), $\alpha = 1$ admits the most expensive feasible provider (full latency optimization unconstrained by cost), and $\alpha \in (0, 1)$ continuously interpolates between the two regimes. Because $B_\alpha(t)$ self-calibrates to

the current envelope of feasible effective costs, α is dimensionless and a uniform sweep gives uniform coverage of the cost–latency Pareto frontier without per-deployment tuning of an absolute budget.

Proposition 3.4 (Sparsity of Optimal Mix)**.** *When the cost budget is tight, the optimal solution uses at most two providers; when the budget is slack, it uses only one.*

This follows from LP basic-feasible-solution theory: a basic solution has at most as many non-zero components as active constraints. The LP has one equality ($\sum_j \pi_j = 1$) and one inequality (the cost budget); when the budget binds it contributes a second active constraint, yielding the two-provider mix. Sparsity is practically valuable: each request is routed to one of at most two providers, simplifying implementation and reducing variance.

Solve Frequency. RouteWise re-solves this LP once per request arrival. The request is then dispatched to a single provider drawn from π: a two-provider mix is realized *across* requests, not by splitting one request. Solving this often is cheap because of Theorem 3.4: the optimum is either a single provider or a two-provider mixture on the budget boundary, therefore RouteWise simply enumerates these closed-form candidates instead of invoking a general LP solver. For the eight-provider pool we evaluate this is arithmetic over a few dozen candidates, negligible beside the network round-trip that follows.

3.3.3 Probability-Targeted Smart Hedging.

Latency-aware routing optimizes the *expected* latency distribution but cannot guarantee strict individual request SLOs. We augment the router with **probability-targeted hedging**, which watches each request in flight and, instead of guessing in advance which one will straggle, repeatedly answers a much easier question: *if I send a backup now, how likely is this request to still meet its deadline?*

Let T_p and T_b denote the TTFT of the primary provider p and candidate backup provider b, modeled by their empirical survival functions $S_p(x) = 1 - F_p(x)$ and $S_b(x) = 1 - F_b(x)$. Suppose a request has been dispatched to the primary and elapsed time t has passed without a response (i.e., $T_p > t$). If we dispatch a backup at time t, the combined probability that at least one request meets the latency SLO L_{SLO} is the complement of both requests failing:

$$P_{\text{succ}}(t) = 1 - \underbrace{\frac{S_p(L_{\text{SLO}})}{S_p(t)}}_{P(T_p > L_{\text{SLO}} \mid T_p > t)} \cdot \underbrace{S_b(L_{\text{SLO}} - t)}_{P(T_b > L_{\text{SLO}} - t)} \tag{7}$$

The hedger asks this question at a grid of checkpoints spaced at fixed fractions of the SLO budget. At each checkpoint it scores every provider available under the same feasibility mask the LP uses, other than the primary, and keeps the *feasible* ones: those that would restore the target reliability, $P_{\text{succ}}(t) \geq 1 - \epsilon$ (e.g., 0.99). Two rules then settle the policy:

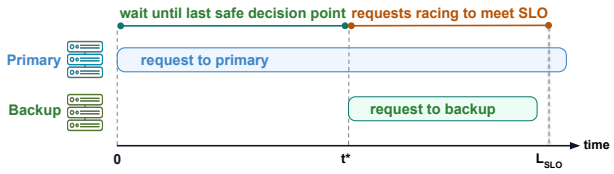


Figure 5. Smart hedging timeline. The primary request is dispatched at t=0. As t grows, the combined primary-plus-backup SLO-success probability $P_{\text{succ}}(t)$ drops. The hedger dispatches a backup at the latest $t^\star$ for which $P_{\text{succ}}(t^\star)$ still meets the target $1 - \epsilon$; the first response is returned.

- **Which backup:** the cheapest feasible one. Feasibility already certifies reliability, so the only thing left to optimize is the price of the duplicate dispatch. That price is the marginal cost of the extra call.
- **When to send it:** at the last checkpoint that still has a feasible backup. Waiting costs nothing if the primary returns on its own, so RouteWise holds back while any later checkpoint would still work and fires at the first one where none would.

The two requests then race. The first token to arrive is forwarded to the user and the losing stream is cancelled, so RouteWise does not pay for the loser's generation. If no checkpoint ever offers a feasible backup, the SLO is unreachable under current conditions and the request rides on the primary as best-effort. Figure 5 illustrates this timeline.

Cost Analysis. When hedging triggers, both requests are billed. The expected effective cost of a routed request under the hedging policy is:

$$\mathbb{E}[C] = c_p + P(\text{hedge}) \cdot c_b \tag{8}$$

where c_p and c_b are the marginal costs of the primary and backup dispatches, and $P(\text{hedge})$ is the probability of a hedge dispatch. Because the backup is dispatched at the latest possible moment, $P(\text{hedge})$ is minimized, keeping expected costs low while preserving the target SLO reliability.

Exploratory Hedging. While our current implementation uses periodic out-of-band probes to track provider latency, high-traffic deployments can eliminate this active probing overhead entirely through *exploratory hedging*. Rather than always routing a hedge to the cheapest feasible backup, the system can periodically sample a random one from the feasible set. Although this may sacrifice a fraction of the backup's SLO-success probability, the resulting hedged requests act as organic exploration probes. At sufficient request volumes, this background rate of hedging naturally yields a dense, up-to-date TTFT profile for all candidate providers, elegantly coupling tail-latency mitigation with continuous system exploration without incurring dedicated probing costs.

4 Evaluation

We evaluate RouteWise to answer the following key questions:

Table 2. Datasets used in evaluation.

				Input tokens			Output tokens			
Workload	Span	Reqs	Total tokens	Mean	Median	P99	Mean	Median	P99	Cache hit (request)
BurstGPT	30 days	1.81M	927M	456	368	1,978	56	14	607	—
FreeInference	7 days	755k	70.8B	93,362	48,646	654,163	506	163	4,548	80%

- How much does RouteWise reduce serving cost relative to state-of-the-art systems and algorithms, and how close does it come to the offline optimal?
- How much does RouteWise reduce latency and SLO violations relative to state-of-the-art systems, and what mechanisms drive these improvements?
- How effectively does RouteWise expose the cost–latency trade-off to users and operators?
- How sensitive is RouteWise's performance to key design choices and parameters, such as output-length prediction and effective cost designs?

4.1 Experimental Setup

Workloads. We evaluate RouteWise on two real-world workloads (Table 2). The first, BurstGPT, is a conversational/API workload from Azure production traces [42]. Because it contains no prompts, we pair its request timestamps with ShareGPT prompts [2], producing a 30-day replay of 1.81M requests. The mean input length is 456 tokens, and prefix caching is disabled. The second workload is a 7-day trace of 754,750 agentic requests from FreeInference [13], our platform providing free LLM inference to the research community. Requests contain 93,362 input tokens on average, and cached prefixes account for 93% of all input tokens.

Provider Ecosystem. We evaluate routing across a candidate pool serving the competitive open-weight model MiniMax-M3 across three complementary provider tiers:

- *Quota subscription*: MiniMax Token Plan ($20/month), providing a depletable quota of 4,500 requests per 5-hour window (or 45,000 requests/week) at zero marginal cost until exhausted.
- *Concurrency subscription*: Featherless ($25/month), offering 2 dedicated concurrent execution slots at zero marginal cost when idle.
- *On-demand metered APIs*: Six commercial endpoints accessed via OpenRouter [33]: Together, GMICloud, MiniMax, AtlasCloud, Novita, and StreamLake. Five charge standard list prices ($0.30/M input, $1.20/M output tokens); GMICloud offers a 20% discount ($0.24/$0.96) with higher latency. Cached tokens are billed at each provider's discounted cached-input rate.

Baselines. We compare RouteWise against the following online baselines.

- **Greedy-cost** routes to the lowest marginal-cost provider and falls back to on-demand APIs when subscription capacity is exhausted. **Greedy-latency** routes to the fastest provider without considering cost.
- **OR-auto**, **OR-latency**, and **OR-price** are three designs provided by OpenRouter [33] that balance latency and cost; the detailed algorithms are not public. We include them in real-world evaluations.
- **Random** chooses uniformly among providers.
- **Single-provider** routes all requests to Together AI, the fastest on-demand provider in our profiling. (supported by OR-latency provider mix in which Together has the largest share)

Metrics. We evaluate all systems using three metrics:

- *Serving Cost*: sum of API charges and fixed subscription fees prorated over the evaluation timeframe. We report it in dollars for the real-provider replay, where it is an actual bill, and normalized to Greedy-cost in simulation.
- *Latency*: mean, median (P50), P95, and P99 TTFT.
- *SLO Violations*: fraction of requests exceeding the TTFT SLO target (L_{SLO}).

Configurations. RouteWise uses three key parameters:

- *Pareto knob (α)*: Normalized cost-budget parameter $\alpha \in [0, 1]$ defining $B_\alpha(t)$ (§ 3.3). $\alpha = 0$ minimizes cost, $\alpha = 1$ greedily minimizes TTFT, and intermediate values trace the Pareto frontier. We use RouteWise-0.5 by default.
- *Latency SLO target (L_{SLO})*: Default TTFT target of 3 s, slightly above nominal 1–2 s provider latency; requests exceeding 3 s are SLO violations. We evaluate sensitivity to $L_{\text{SLO}} \in \{2, 3, 4, 5\}$ s in § 4.4.2.
- *Smart hedging*: Hedging checks in-flight requests and races at most one backup when success probability drops below target reliability ($1 - \epsilon = 0.99$), canceling the loser on first token. Enabled by default across all RouteWise configurations.

Implementation and Testbeds. We evaluate RouteWise across two environments:

- *Real-world prototype testbed*: Implemented in Python and evaluated on a server with an AMD Ryzen Threadripper PRO 7975WX CPU (32 cores / 64 threads), 512 GB DRAM, and a 25 Gbps NIC (network RTT $<$ 50 ms to all endpoints).
- *Trace-driven simulator*: long-horizon workloads such as the 30-day BurstGPT trace are impractical to replay against live providers: doing so is costly, and provider availability can change over the course of an experiment

as models or endpoints are retired. Live experiments also do not allow us to control provider behavior, which is necessary for testing robustness and evaluating design choices under different performance conditions. We therefore built a simulator that replays request traces.

4.2 Real-world Experiments

Execution and replay methodology. We evaluate our prototype by replaying a 24-hour segment of the BurstGPT trace (14,233 requests per policy) against live endpoints for MiniMax-M3. Because provider latency varies dynamically with backend load, evaluating policies sequentially would introduce temporal bias. We therefore ran all eleven policies—the six baselines and five RouteWise operating points ($\alpha \in \{0, 0.25, 0.5, 0.75, 1\}$)—concurrently within the same 24-hour window, launched one minute apart, sharing the live provider pool and telemetry profiler. Each replayed prompt carried a unique prefix to isolate prefix caches across policies. Reported costs reflect actual metered API charges plus the prorated 24-hour subscription fees.

4.2.1 E2E Performance and Baseline Comparison.

Figure 1 compares RouteWise against six baseline policies on the 24-hour live MiniMax-M3 replay across cost, mean TTFT, and tail SLO violations.

Comparison with latency-oriented baselines. RouteWise achieves competitive or superior latency while substantially reducing serving cost. Relative to Greedy-latency, RouteWise-1 achieves lower mean TTFT (0.70 s versus 0.81 s, a 13.6% reduction) at comparable cost ($3.25 versus $3.28). Compared with OpenRouter's latency-oriented mode (OR-latency) and Single-provider, RouteWise strictly dominates both: even the balanced configuration RouteWise-0.5 achieves lower mean TTFT (0.74 s versus 0.81 s and 0.83 s) while reducing cost by 9.7% and 8.8% ($2.91 versus $3.23 and $3.20), respectively. Across all evaluated policies, RouteWise-1 achieves the lowest mean TTFT, outperforming the fastest baseline by 12.9%.

Comparison with cost-oriented baselines. RouteWise matches the cost efficiency of low-cost baselines while dramatically improving latency. Relative to OR-price, the cheapest OpenRouter mode, RouteWise-0 incurs essentially identical cost ($2.46 versus $2.44, within 1%) while reducing mean TTFT from 1.30 s to 0.90 s (30.8%). Compared with Greedy-cost ($1.97), RouteWise-0 spends 24% more ($2.46), but reduces mean TTFT from 1.02 s to 0.90 s (11.8%). Furthermore, relative to OpenRouter's default multi-provider mode (OR-auto, $3.03, 1.18 s), RouteWise-0 strictly dominates it across both dimensions, cutting cost by 18.9% and mean TTFT by 23.2%. Of that $2.46, $0.96 is metered spend and $1.50 the prorated subscription fees, whereas OR-auto's $3.03 is metered throughout.

SLO compliance. Using a 3-second TTFT SLO target, every baseline violates the SLO on 1.6%–3.3% of requests: 3.1%–3.3% for cost-oriented policies (OR-price, OR-auto, and Greedy-cost), and 1.6%–2.1% even for latency-oriented baselines (Single-provider, OR-latency, and Greedy-latency). In contrast, RouteWise bounds the violation rate between 0.50% and 0.83% across all operating points—a 2×–6× reduction compared to any baseline.

Navigating the Pareto frontier via α. Beyond matching or dominating individual baselines, Figure 1 confirms that α provides operators with a continuous, predictable lever to trade cost for latency. Sweeping α from 0 to 1 reduces mean TTFT from 0.90 s to 0.70 s (and median TTFT from 0.74 s to 0.50 s, Figure 6a), as serving cost scales monotonically from $2.46 to $3.25. Intermediate values ($\alpha \in \{0.25, 0.5, 0.75\}$) yield smooth speedups (0.83 s, 0.74 s, and 0.72 s at $2.64, $2.91, and $3.13). Across the entire sweep, P99 TTFT stays below 2.85 s, demonstrating that the knob trades cost for body latency without sacrificing the 3-second SLO.

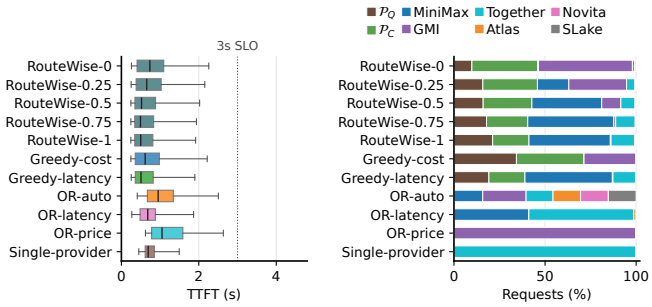
Provider mix dynamics. This Pareto frontier is realized by dynamically restructuring the provider mix according to the operator's cost budget $B_\alpha(t)$ (Figure 7, Figure 6b). In our provider pool, on-demand APIs present a stark trade-off: five providers charge the standard list price ($0.30/$1.20 per million input/output tokens) but differ by up to 2× in TTFT—MiniMax and Together average 0.72–0.84 s, whereas AtlasCloud and StreamLake reach 1.5–1.6 s (Figure 7a). GMICloud provides a 20% discount (Figure 7b) but is among the slowest (1.17 s). Meanwhile, the two subscriptions provide low-latency, zero-marginal-cost capacity (0.72 s for concurrency $\mathcal{P}_C$, 0.82 s for quota $\mathcal{P}_Q$), bounded by concurrency caps and daily quotas.

As α increases, the LP mixer translates the relaxed cost budget into a systematic reallocation of traffic (Figure 6b). RouteWise-0 absorbs 47% of all requests via zero-marginal-cost subscriptions (37% concurrency, 10% quota) and routes nearly all on-demand overflow (52% of total requests) to GMICloud, the cheapest API. As α increases, the router progressively substitutes GMICloud with faster endpoints: GMICloud's share collapses to below 1% with RouteWise-1, replaced by premium on-demand providers (MiniMax growing to 45%, Together to 13%) and heavier use of the quota subscription (21%). This progressive promotion of requests into lower-latency tiers directly drives the end-to-end TTFT reductions, demonstrating that α reliably controls the system's operational regime.

4.2.2 Sources of Performance Advantage.

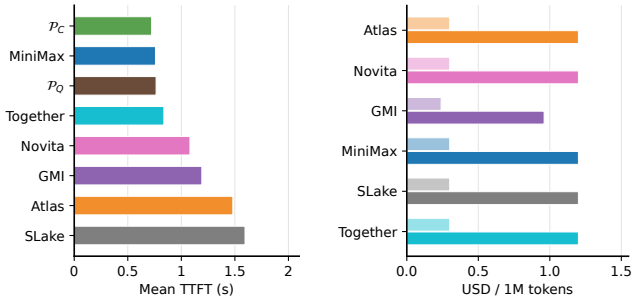
RouteWise's gains come from expanding multi-provider routing beyond on-demand APIs and then exploiting the resulting resource diversity. Three mechanisms explain the advantage.

Expanding the serving pool beyond on-demand APIs. Existing multi-provider routers choose among metered API endpoints. RouteWise additionally incorporates quota- and concurrency-based subscriptions, creating a more diverse pool in both provider performance and pricing structure.



(a) TTFT distributions **(b)** Provider mix

Figure 6. TTFT distributions and provider mix in the 24-hour MiniMax-M3 replay. Boxes span the 25th–75th percentiles, with medians marked; whiskers span the 5th–95th percentiles.



(a) Provider latency **(b)** Provider pricing

Figure 7. Provider latency and token prices in the MiniMax-M3 replay. Light and dark bars show input and output prices, respectively.

This introduces zero-marginal-cost capacity while also adding alternative endpoints that can be useful when on-demand providers are slow.

Allocating heterogeneous capacity where it is most valuable. Subscription capacity cannot be treated like another on-demand endpoint: concurrency slots are temporarily occupied, while quotas are cheap but depletable. RouteWise explicitly models these constraints and dynamically prices scarce quota, preserving it for requests that would otherwise be expensive to serve on demand. Thus, the benefit comes not only from having subscription capacity, but from spending it where it provides the greatest value.

Rescuing latency outliers that routing cannot anticipate. Even with a diverse provider pool and up-to-date latency profiles, individual requests can still become unpredictable stragglers. RouteWise complements *a priori* routing with selective in-flight hedging, launching a backup only when the primary becomes unlikely to meet its SLO. This truncates latency tails without the cost of indiscriminately duplicating requests.

In short, RouteWise gains from a broader serving pool, resource-aware allocation within that pool, and in-flight recovery when the initial routing decision goes wrong.

4.2.3 Effectiveness of System Components.

To isolate the mechanisms driving these end-to-end gains, we evaluate the three core components of RouteWise: dynamic LP traffic rebalancing under shifting provider latencies, subscription capacity management across depletable quotas and regenerating concurrency slots, and probability-targeted hedging.

LP rebalancing. The LP re-solves for every request against the rolling latency profile, so the provider mix should follow provider latency rather than a fixed preference. Figure 8 shows this behavior for RouteWise-0.5. As GMICloud's rolling TTFT roughly doubles over the run, from about 1s to about 1.9s, the LP walks its share down from a peak of 23% to zero, and it hands traffic back to the Featherless slot once that provider recovers from a spike above 4s. Every provider's assigned share moves opposite to its own latency. This is re-weighting rather than failover: GMICloud never

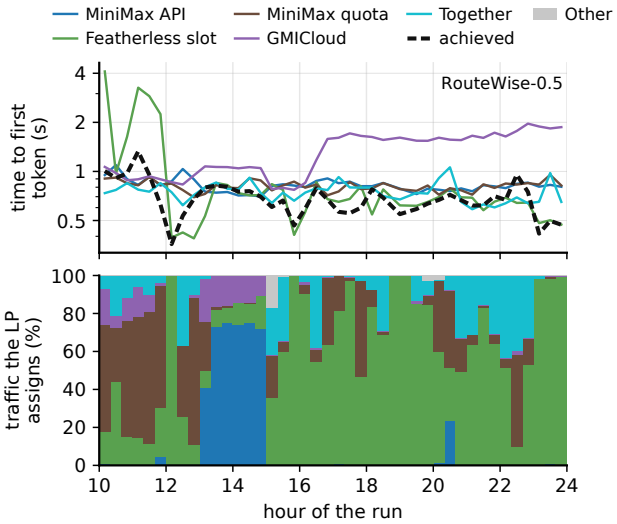


Figure 8. LP rebalancing during the 24-hour replay. Top: rolling latency estimates used by the LP and achieved mean TTFT (dashed). Bottom: average routing weights. Results are grouped into 20-minute bins, with sparse bins merged.

failed or hit a rate limit, and the only availability that moves is the concurrency slot's, which is offered only while a slot is free (Table 3).

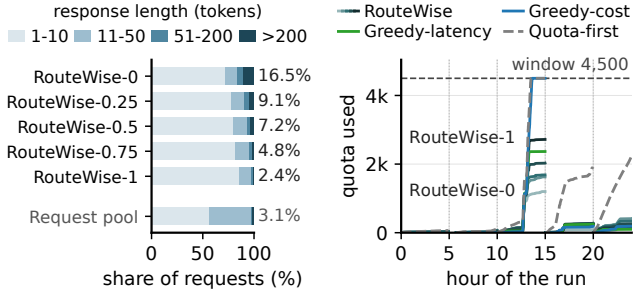
Quota effective cost. The exponential effective cost in Equation 3 helps RouteWise preserve quota for requests that would otherwise be expensive to serve on demand (Figure 9). For RouteWise-0, this makes quota admission highly selective. When the concurrency slot is busy, requests longer than 50 tokens make up 16.5% of quota admissions, even though they are only 3.1% of the request pool those admissions are drawn from (Figure 9a). Since such long responses are relatively rare in this workload, RouteWise uses only 27% of the 4,500-request quota window (1,458 requests), leaving substantial capacity unused rather than spending it on cheaper requests (Figure 9b). As α increases, this selectivity gradually relaxes: the router admits more requests to the quota endpoint, but those requests are shorter on average.

Table 3. Concurrency-slot handling on the replay. RouteWise prioritizes free capacity but trades it for latency as α increases. Available slots have zero cost, so they are always considered (100% offered) and chosen when fastest. When slower than metered providers, the take rate drops from 100% to 0% to buy speed, reducing the slot's request share from 37% to 20%.

	Free-slot	Offered	Taken when		
α	decisions	when free	fastest	not	Share
0	5,300	100%	100%	100%	37%
0.25	5,490	100%	100%	57%	30%
0.5	5,840	100%	100%	34%	27%
0.75	8,355	100%	100%	7%	23%
1	9,023	100%	100%	0%	20%

Table 4. Smart hedging on the 24-hour MiniMax-M3 replay with 3-second SLO. Racing backups for at-risk queries (5.2%–7.8% hedged, winning ≈50% of races) truncates tail latency, reducing SLO violations from 2.0%–3.1% to 0.5%–0.8% and cutting P99 TTFT by up to 64% at a modest 5.4%–11.8% extra metered cost. "With / without" compares against the counterfactual where hedged requests waited solely for the primary leg. Extra cost reflects token charges incurred by losing legs.

			With / without hedging			
α	Hedged	Wins	Violations	Mean TTFT (s)	P99 TTFT (s)	Extra cost
0	7.8%	52%	0.83% / 3.11%	0.90 / 1.11	2.84 / 7.96	11.8%
0.25	7.6%	48%	0.50% / 2.68%	0.83 / 1.00	2.68 / 6.01	10.8%
0.5	6.4%	47%	0.61% / 2.32%	0.74 / 0.89	2.68 / 5.51	7.8%
0.75	5.9%	48%	0.50% / 2.36%	0.72 / 0.85	2.65 / 4.98	6.3%
1	5.2%	46%	0.54% / 1.98%	0.70 / 0.80	2.72 / 4.37	5.4%



(a) What quota is spent on **(b)** Quota used over the day

Figure 9. RouteWise reserves quota for requests that are expensive to serve on demand. (a) Response-length distribution of quota-admitted requests versus the eligible request pool, where the quota competes with metered providers. Under a tight budget, quota disproportionately serves long responses; this selectivity decreases as α increases. (b) This selective allocation keeps RouteWise within the five-hour quota limit, while Greedy-cost exhausts it early and falls back to metered providers.

RouteWise-1 admits 3,113 requests, using 60% of the window. Greedy-cost shows the opposite behavior: it consumes quota without distinguishing between requests, exhausts the window 13.6 hours into the day, and then serves the rest of the day at metered prices.

Concurrency effective cost. Because an idle slot is wasted, RouteWise prices an available slot at zero, which should make it a candidate whenever it is free. Table 3 confirms both halves of that behavior: at every operating point the slot entered the candidate set in 100% of the decisions in which it was free, and was taken in essentially all of the decisions in which it was also the fastest candidate. The last two columns show the cost budget's pull. When the slot was free but slower than some metered provider, RouteWise-0 still took it every time, while the take rate falls to 57%, 34%, 7% and 0% as α rises and the budget allows paying for speed. Its overall share of requests therefore falls from 37% to 20%, even though the slot sits idle more often at high α.

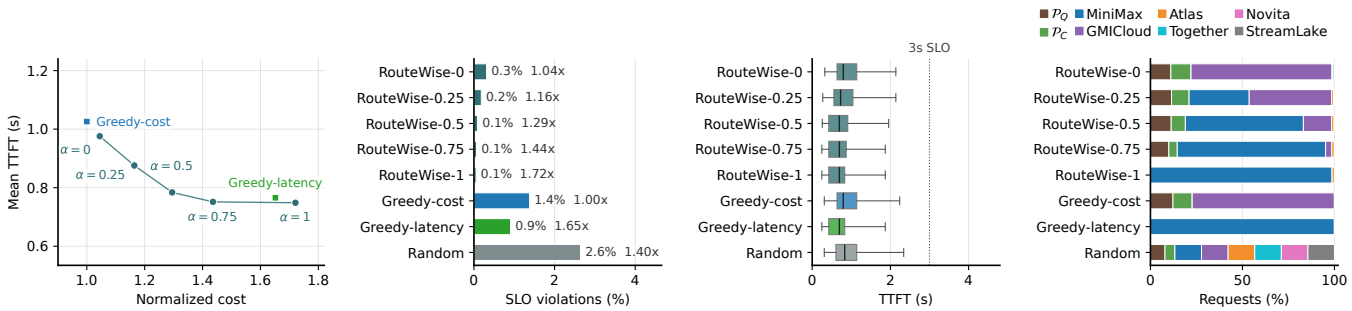
Hedging. Table 4 isolates hedging's contribution at each operating point. Hedging fires on only 5–8% of requests and the backup wins about half of its races, yet those few races remove most of the tail: without hedging, violations would be 2.0–3.1% rather than 0.5–0.8%, and P99 TTFT 4.4–8.0 s rather than 2.7–2.8 s. Because the rescued requests are the slowest ones, cutting the tail also lowers mean TTFT by 12–19%. The price is modest: the losing legs add an estimated 5–12% to metered spend, and the share falls with α as faster primaries need fewer backups.

4.3 End-to-end Experiment in Simulation

We use the trace-driven simulator to explore more settings: a month-long horizon and a workload whose request shape and pricing differ sharply from BurstGPT's. Both replays reuse the testbed's provider pool (§ 4.1), draw provider TTFT from the 202k latency samples collected during the 24-hour run, prorate the monthly subscriptions over the replay, and keep the default 3-second SLO; OpenRouter's proprietary modes cannot be executed offline, so the baselines are Greedy-cost, Greedy-latency, and Random. Because those latency distributions come from the run behind § 4.2, agreement with it is not evidence of simulator fidelity.

4.3.1 30-day BurstGPT Workload. Replaying all 1.81M requests (127× the live replay) leaves the picture of § 4.2 intact at 30× the horizon (Figure 10): the same frontier, from 0.98s mean TTFT at 1.04× Greedy-cost (RouteWise-0) to 0.75s at 1.72× (RouteWise-1); RouteWise-0.75 dominating Greedy-latency outright (0.75s versus 0.77s, 13% lower cost, 9× fewer violations); violations of 0.1–0.3% across the sweep against 0.9–2.6% for the baselines.

Worth carrying forward is the shape of the provider mix (Figure 10d): MiniMax's share rises in step with α (33%, 64%, 81%, 99%), GMICloud gives up the complement, and the two capacity-bound subscriptions hold about a fifth of the month throughout. That ramp is the budget arithmetic of Equation 5: with only the metered APIs feasible, separated by GMICloud's 20% discount, a budget a fraction α of the way from the cheapest to the priciest admits the faster, pricier



(a) Cost–TTFT frontier **(b)** SLO violation **(c)** TTFT distributions **(d)** Provider mix

Figure 10. Thirty-day BurstGPT simulation. Costs in (a) and the × labels in (b) are normalized to Greedy-cost. Boxes show the interquartile range and median; whiskers span the 5th–95th percentiles.

endpoint for about that fraction of requests. The next workload breaks this.

4.3.2 FreeInference Agentic Workload. An increasing share of production traffic is agentic rather than conversational: multi-turn, long-context, and heavy on prefix-cache reuse [11, 17, 40]. We replay the 7-day FreeInference trace (754,750 requests) from FreeInference on the same pool, with prefix caching enabled and cached tokens billed at each provider's discounted cached-input rate.

The comparisons carry over (Figure 11): RouteWise-0.25 costs 19% less than Greedy-latency at a lower mean TTFT (0.75s versus 0.77s) with 9× fewer SLO violations (0.1% versus 0.9%), and RouteWise-0, within 2% of Greedy-cost's spend, cuts violations from 1.3% to 0.1%. The mechanisms split the work as in § 4.2.3: the LP balances the on-demand mix within the budget, the dynamic quota effective cost reserves the free tier for the longest prompts, and hedging trades a little metered spend for the tail. What differs is that the subscriptions barely participate: the 45,000-requests-per-week quota and the two concurrency slots absorb under 9% of the 755k requests, so the on-demand mix sets the trade-off.

4.4 Ablation Study

4.4.1 Offline Analysis. We use an offline oracle as a diagnostic lower bound to evaluate the efficiency of RouteWise's prepaid capacity allocation on the 30-day BurstGPT replay. Quota and concurrency exhibit distinct online-to-offline gaps due to their differing semantics. For depletable quota, RouteWise costs 15.0% above the offline lower bound, versus 19.3% for Greedy-cost. For regenerating concurrency, both policies use zero-cost admission and finish 6.0% above offline. These cost gaps establish the outcome but do not, by themselves, prove the allocation mechanism.

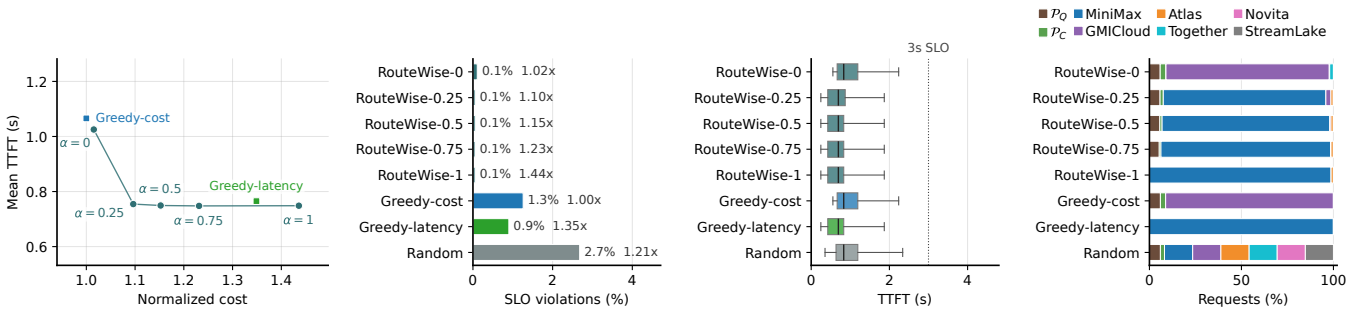
4.4.2 SLO Target. The SLO is an operator knob, but a tighter target does not simply buy lower latency. Table 5 evaluates RouteWise-0.5 at four targets with the same real world experiment setup. Relaxing the target from 2 s to 5 s cuts violations from 8.5% to 0.4% at nearly constant cost, because the router pays for a tight target in hedging rather than in money: the hedged share falls from 14.6% to 4.8%. Mean TTFT, however, is lowest at 3 s. Hedging is a safety net rather than a first-line strategy: it waits until a request appears unable to meet its deadline and dispatches at most one backup. Under a target the pool cannot reliably meet, far more requests reach that point, the backup wins only a third of its races because a deadline the primary cannot meet is often one the backup cannot meet either, and the elapsed wait plus the backup's own tail leave these requests deep in the distribution, dragging the mean up. A very loose target instead hedges too rarely to rescue slow requests, and mean TTFT drifts up again. The target should therefore sit slightly above what the pool typically delivers, which motivates our 3-second default.

Table 5. SLO-target sweep. RouteWise-0.5 with hedging replayed the same 24-hour trace at four TTFT targets, concurrently in a separate window; each run is scored against its own target.

SLO	Cost ($)	Mean (s)	P99 (s)	Viol.	Hedged	Wins
2 s	2.64	1.02	6.10	8.5%	14.6%	32%
3 s	2.62	0.90	4.01	2.2%	13.2%	49%
4 s	2.60	0.91	3.74	0.8%	7.7%	53%
5 s	2.60	0.94	4.35	0.4%	4.8%	66%

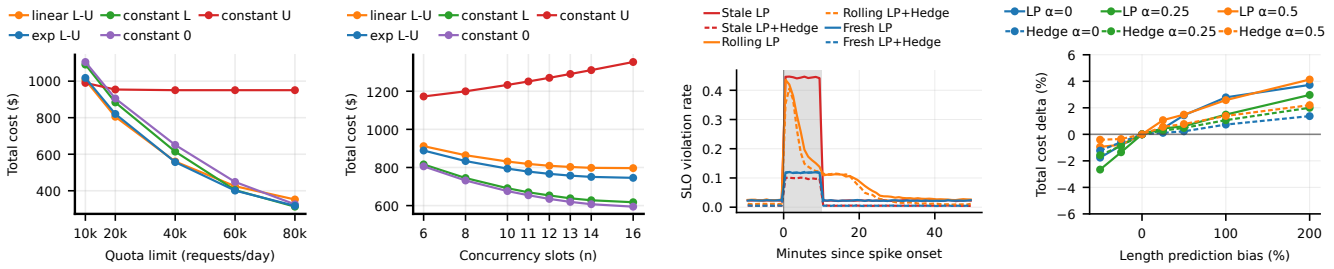
4.4.3 Effective Cost Formula. We evaluate alternative effective cost formulations for subscription capacity. For quota providers, our dynamic exponential formula performs best on average across diverse workloads (Figure 12a), though static pricing can win in isolated configurations if the static price perfectly matches the quota's scarcity. Because production workloads are highly bursty, it is impossible to know a quota's true scarcity in advance. The exponential formula is robust because it self-corrects: staying cheap during underutilization to maximize return, and ramping up aggressively when capacity tightens.

Conversely, for concurrency subscriptions, assigning a fixed cost of zero strictly outperforms dynamic pricing (Figure 12b). This is intuitively sound: unlike a monthly quota that depletes permanently, concurrency slots instantly regenerate when a request completes. If an available slot is not used immediately, that capacity window is permanently



(a) Cost–TTFT frontier **(b)** SLO violation **(c)** TTFT distributions **(d)** Provider mix

Figure 11. Seven-day FreeInference simulation with prefix caching enabled. Costs in (a) and the × labels in (b) are normalized to Greedy-cost for this workload. Boxplot conventions follow Figure 10.



(a) Quota effective cost **(b)** Concurrency effective cost **(c)** Latency misprediction **(d)** Output-length misprediction

Figure 12. Ablation studies on the 30-day BurstGPT replay. (a) Dynamic exponential quota pricing outperforms linear and static formulas. (b) A constant-zero effective cost dominates for concurrency slots. (c) Robustness to latency misprediction across three same-price providers under a 2-second SLO. Every hour, the fastest provider experiences a 3× TTFT spike for 10 minutes (shaded). Solid lines use LP routing alone, while dashed lines add hedging. Colors indicate the router's latency knowledge: stale (never updated), the production 15-minute rolling profile, or fresh (oracle). (d) Robustness to output-length misprediction: a +200% estimation bias increases total cost by at most 4.1%.

wasted. Thus, a greedy zero-cost design safely maximizes the utilization of rapidly regenerating concurrency resources before incurring any metered API costs.

4.4.4 Robustness to Latency Misprediction. When provider latency spikes suddenly, background telemetry updates lag behind reality, causing *latency misprediction* where the LP router continues dispatching requests based on outdated statistics. Figure 12c evaluates this scenario in simulation by increasing the fastest provider's TTFT by 3× for 10 minutes every hour across a 30-day BurstGPT replay (three same-price providers, 2-second SLO). Under stale telemetry, routing with the LP alone causes SLO violations to jump from 2.3% to 44%.

Smart hedging acts as an orthogonal safeguard against latency misprediction. Because hedging triggers on in-flight elapsed time rather than historical telemetry, it directly detects straggling requests without waiting for profiler updates. Adding hedging to stale beliefs caps violations near 10%. *Fresh* telemetry is an idealized oracle whose latency estimates update the instant a provider's TTFT changes, an upper bound on what faster profiling alone can achieve. Hedging with stale beliefs outperforms even a fresh oracle LP without hedging (12% violations), as racing providers is more effective than abandoning the primary entirely. Profiler freshness and runtime hedging are therefore complementary.

4.4.5 Robustness to Output-Length Misprediction. While stale telemetry introduces estimation error on latency, unknown response length introduces uncertainty on cost: the router must predict generated tokens $\hat{n}_{\text{out}}$ upfront to compute on-demand effective costs. We evaluate RouteWise's sensitivity to this error by injecting prediction bias in $[-50\%, +200\%]$ on the 30-day BurstGPT replay (Figure 12d).

Cost is insensitive to output-length misprediction because generated responses are short compared to inputs across diverse workloads. In BurstGPT, median output length is 14 tokens versus 368 input tokens (Table 2); in FreeInference, context is even more asymmetrical (mean 93,362 input versus 506 output tokens). For instance, in a typical request with a 100K cached prefix, 2K new input, and 200 output tokens, output accounts for only 3% of total serving cost. Because output tokens represent such a small cost fraction, estimation errors yield small shifts in effective costs. Our ablation confirms that even a +200% prediction error raises total cost by only 1.4%–4.1% across operating points.

5 Discussion

Provider Quality Equivalence. Our framework assumes all providers serving the same open-weight model, such as MiniMax-M3, return equivalent outputs. This holds when weights are identical and sampling parameters are passed

through. In practice, some providers apply FP8 quantization, which introduces small perplexity differences, or provider-side safety filters, which alter refusal rates. Extending the framework to quality-aware routing, where providers differ in both cost and output fidelity, is a natural direction.

Generalization to Local GPU Deployments. Although our examples focus on public API providers, the same resource types arise in self-hosted settings. A GPU cluster with k replicas maps to a concurrency-limited provider in $\mathcal{P}_C$, and a gutter pool of hot-standby nodes maintained for high availability [14, 20] maps to a capacity-limited quota provider in $\mathcal{P}_Q$. The routing algorithms apply unchanged; only capacity profiling differs.

Adversarial Arrivals. The competitive ratio guarantee of Theorem 3.3 holds for worst-case sequences, but the exponential threshold is tuned for stochastic workloads. An adversary that front-loads all high-value requests can cause the effective cost to rise too slowly. In practice, LLM workloads are not adversarial, and our experiments confirm near-optimal performance across diverse real-world traces.

6 Related Work

LLM Routing and Cascading. Model routers balance inference cost and response quality. FrugalGPT [6] learns model cascades, RouteLLM [30] learns routing from preference data, and AutoMix [1] uses self-verification to decide whether to query a larger model. Other work combines routing and cascading [10] or assigns queries across models under batch-level cost and capacity constraints [24]. RouteWise fixes the model and selects its provider; a model router could use RouteWise after choosing a model. OpenRouter [32, 35] supports provider selection, fallbacks, and price and performance preferences. RouteWise focuses on allocating requests across prepaid quotas, concurrency slots, and metered APIs, accounting for their different capacity constraints and costs. Subscription capacity couples decisions across time. Consuming a quota slot now changes the effective cost of later requests, even when their prompts and the providers' posted prices are identical.

Multi-Cloud Resource Management. SkyPilot [44] brokers compute resources across clouds. SkyServe [23] combines spot and on-demand replicas across regions and clouds, while SpotServe [26] adapts LLM execution to GPU preemptions. Can't Be Late [43] uses on-demand instances as a fallback to meet job deadlines despite spot interruptions. These systems reduce costs by managing infrastructure and model replicas. RouteWise operates over existing API endpoints with subscription capacity. It decides how to use remaining quota and available concurrency alongside APIs, without provisioning compute resources or migrating replicas.

LLM Serving and Cache-Aware Scheduling. Orca [45] and vLLM [21] improve batching and memory management. DistServe [48] separates prefill and decoding, while Llumnix [39] reschedules requests across instances through live migration. Prefix reuse also affects placement: SGLang [47] reuses KV state through RadixAttention, and Preble [38] jointly considers prefix reuse and load balancing when scheduling requests across GPUs. These systems control execution or cache placement within a serving deployment. RouteWise has no direct control over provider KV caches. It incorporates estimated cached tokens and provider-specific cached-input prices into effective costs, while using observed TTFT to guide latency decisions. A cache discount changes the monetary comparison without implying that an endpoint will meet the latency SLO; provider queueing can still dominate the time to the first token.

Tail Latency and Hedging. Hedged requests reduce tail latency by sending a delayed duplicate and using the first response [8]. Delaying duplicates until the primary exceeds a latency percentile limits the added load. Primorac et al. [36] show that redundant work can worsen congestion and propose combining load balancing with work-conserving hedging. RouteWise applies hedging across providers with different prices and TTFT distributions. It estimates whether a backup can restore the target probability of meeting the TTFT SLO, delays dispatch to the last feasible checkpoint, and selects the cheapest feasible backup. This requires considering the primary's elapsed waiting time together with each backup's latency distribution. An endpoint that could meet the SLO at arrival may be too slow to serve as a late backup.

Online Resource Allocation. Online knapsack algorithms [16, 49] and primal-dual methods [4] allocate finite capacity without knowing future demand. RouteWise's quota threshold draws on this literature, valuing a request by the metered cost that quota would avoid. Ski-rental algorithms [19, 46] study when to rent or buy; our subscriptions are already purchased, so decisions concern how to use their capacity. Qiu et al. [37] predict output lengths with a proxy model to prioritize shorter requests and reduce head-of-line blocking. RouteWise uses a lightweight bucket-mean estimate to compare expected API charges with subscription effective costs and evaluates sensitivity to estimation errors. The estimate supports a billing comparison at dispatch and does not require control over a provider's execution order.

7 Conclusion

We present RouteWise, a cost- and latency-aware routing system for multi-provider LLM serving across heterogeneous pricing schemes, including on-demand APIs and capacity-constrained subscriptions. Compared with state-of-the-art LLM routers, RouteWise reduces cost by 18.9%, latency by 23.2%, and SLO violations by 74.6%, and exposes a single cost knob for navigating the cost–latency trade-off.

References

[1] Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. 2025. AutoMix: Automatically Mixing Language Models. arXiv:2310.12963 [cs.CL] https://arxiv.org/abs/2310.12963

[2] anon8231489123. 2023. ShareGPT Vicuna Unfiltered Dataset. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.

[3] Arli AI. 2026. Arli AI Pricing. https://www.arliai.com/pricing.

[4] Niv Buchbinder and Joseph (Seffi) Naor. 2009. The Design of Competitive Online Algorithms via a Primal–Dual Approach. *Foundations and Trends in Theoretical Computer Science* (2009). doi:10.1561/0400000024

[5] Cerebras. 2026. Cerebras Code Pricing. https://www.cerebras.ai/pricing.

[6] 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* (2024).

[7] Chutes. 2026. Pricing. https://chutes.ai/pricing.

[8] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[9] DeepSeek. 2026. DeepSeek V4 Preview Release. https://api-docs.deepseek.com/news/news260424.

[10] Jasper Dekoninck, Maximilian Baader, and Martin Vechev. 2025. A Unified Approach to Routing and Cascading for LLMs. arXiv:2410.10347 [cs.CL] https://arxiv.org/abs/2410.10347

[11] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. 2025. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? arXiv:2509.16941 [cs.SE]

[12] Featherless.ai. 2025. Featherless.ai Pricing. https://featherless.ai/docs/plans.

[13] FreeInference. 2026. FreeInference: An OpenAI-Compatible API Powered by Frontier Open Models. https://freeinference.org/. Accessed: 2026-09-19.

[14] Google SRE. 2016. Introduction - Site Reliability Engineering. https://sre.google/sre-book/introduction/.

[15] Gurobi Optimization, LLC. 2024. Gurobi Optimizer Reference Manual. https://www.gurobi.com.

[16] Xin Han, Yasushi Kawase, and Kazuhisa Makino. 2015. Randomized algorithms for online knapsack problems. *Theoretical Computer Science* 562 (2015), 395–405. doi:10.1016/j.tcs.2014.10.017

[17] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations*.

[18] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361* (2020).

[19] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. 1994. Competitive Randomized Algorithms for Non-Uniform Problems. *Algorithmica* 11, 6 (1994), 542–571. https://courses.csail.mit.edu/6.895/fall03/handouts/papers/karlin.pdf.

[20] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. 2024. Revisiting Reliability in Large-Scale Machine Learning Research Clusters. *arXiv preprint arXiv:2410.21680* (2024).

[21] Woosuk Kwon et al. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:2309.06180. https://arxiv.org/pdf/2309.06180.

[22] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christopher Cosgrove, Christopher D. Manning, Christopher Re, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. 2023. Holistic Evaluation of Language Models. *Transactions on Machine Learning Research* (2023).

[23] Ziming Mao, Tian Xia, Zhanghao Wu, Wei-Lin Chiang, Tyler Griggs, Romil Bhardwaj, Zongheng Yang, Scott Shenker, and Ion Stoica. 2025. SkyServe: Serving AI Models across Regions and Clouds with Spot Instances. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 159–175. doi:10.1145/3689031.3717459

[24] Jelena Markovic-Voronov, Kayhan Behdin, Yuanda Xu, Zhengze Zhou, Zhipeng Wang, and Rahul Mazumder. 2026. Robust Batch-Level Query Routing for Large Language Models under Cost and Capacity Constraints. arXiv:2603.26796 [cs.LG]

[25] Meta AI. 2024. Llama 3.3 Model Cards and Prompt Formats. https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/.

[26] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2023. SpotServe: Serving Generative Large Language Models on Preemptible Instances. arXiv:2311.15566 [cs.DC] https://arxiv.org/abs/2311.15566

[27] MLOps Community. 2023. LLM Survey Report 2023. https://mlops.community/wp-content/uploads/2023/06/llm-survey-report.pdf.

[28] Moonshot AI. 2026. Kimi K2.6. https://platform.moonshot.ai/.

[29] Ollama. 2026. Ollama Cloud Pricing. https://ollama.com/pricing.

[30] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. 2025. RouteLLM: Learning to Route LLMs from Preference Data. In *International Conference on Learning Representations*.

[31] OpenRouter. 2026. MiniMax M2.5 on OpenRouter. https://openrouter.ai/minimax/minimax-m2.5.

[32] OpenRouter. 2026. Model Fallbacks. https://openrouter.ai/docs/guides/routing/model-fallbacks. Accessed: 2026-05-14.

[33] OpenRouter. 2026. OpenRouter: The Unified Interface for LLMs. https://openrouter.ai/.

[34] OpenRouter. 2026. Prompt Caching. https://openrouter.ai/docs/features/prompt-caching.

[35] OpenRouter. 2026. Provider Routing. https://openrouter.ai/docs/guides/routing/provider-selection. Accessed: 2026-09-20.

[36] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. 2021. When to Hedge in Interactive Services. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 373–387. https://www.usenix.org/conference/nsdi21/presentation/primorac

[37] Haoran Qiu, Weichao Mao, et al. 2024. Efficient Interactive LLM Serving with Proxy Model-based Sequence Length Prediction. arXiv:2404.08509. https://arxiv.org/pdf/2404.08509.

[38] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2025. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=meKEKDhdnx

[39] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large

Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 173–191. https://www.usenix.org/conference/osdi24/presentation/sun-biao

[40] Muxin Tian, Zhe Wang, Zhenwei Tang, Blair Yang, Kunlun Zhu, Honghua Dong, Hanchen Li, Xinni Xie, Guangjing Wang, and Jiaxuan You. 2026. SWE-Bench Mobile: Can Large Language Model Agents Develop Industry-Level Mobile Applications?. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '26)*. Association for Computing Machinery, 8077–8087. doi:10.1145/3770855.3818488

[41] Together AI. 2026. Together AI Pricing. https://www.together.ai/pricing.

[42] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. arXiv:2401.17644 [cs.DC] https://arxiv.org/abs/2401.17644

[43] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric J. Friedman, Scott Shenker, and Ion Stoica. 2024. Can't Be Late: Optimizing Spot Instance Savings under Deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, Laurent Vanbever and Irene Zhang (Eds.). USENIX Association, 185–203. https://www.usenix.org/conference/nsdi24/presentation/wu-zhanghao

[44] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. 2023. SkyPilot: An Intercloud Broker for Sky Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 437–455. https://www.usenix.org/conference/nsdi23/presentation/yang-zongheng

[45] Gyeong-In Yu et al. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. https://www.usenix.org/system/files/osdi22-yu.pdf.

[46] Keyuan Zhang, Zhongdong Liu, Nakjung Choi, and Bo Ji. 2024. Learning-augmented Online Algorithm for Two-level Ski-rental Problem. arXiv:2402.06715 [cs.DS] https://arxiv.org/abs/2402.06715

[47] Lianmin Zheng et al. 2023. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104. https://arxiv.org/pdf/2312.07104.

[48] Yinmin Zhong et al. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*. https://www.usenix.org/system/files/osdi24-zhong-yinmin.pdf.

[49] Yunhong Zhou, Deeparnab Chakrabarty, and Rajan Lukose. 2008. Budget Constrained Bidding in Keyword Auctions and Online Knapsack Problems. In *Internet and Network Economics*. Springer, 566–576. doi:10.1007/978-3-540-92185-1_63