GoodServe: GLOBALLY-COORDINATED ROUTING OF AGENTIC LLM INFERENCES FOR HIGH GOODPUT

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic applications consume LLM inferences as complete outputs: a code patch or a database query is useless until its last token arrives, so what matters to the application is end-to-end latency against a deadline, and what matters to the operator is *goodput*, the number of inferences that finish within their deadlines. Operators usually serve these inferences on pools of heterogeneous GPUs, and every request must be routed to one instance on arrival. Existing routers are greedy: each request goes to the instance that would serve it fastest or is least loaded, which spends the strongest GPUs on requests that would have met their deadlines on weaker ones and leaves urgent requests waiting. Goodput instead calls for *globally coordinated* routing, in which a request may be placed sub-optimally for itself so that the pool meets more deadlines overall. Coordination needs two quantities a greedy router never estimates: how long each request will run, since its output length is unknown until it finishes, and how fast each instance currently is. *GoodServe* formulates routing as a global goodput problem whose latency model couples both, including prefix-cache reuse, and solves it in a *predict-and-rectify* manner: a mixture-of-experts predictor gated by task type estimates output length accurately and efficiently; an EMA-smoothed black-box estimator tracks each instance's queuing delay and per-token speed; a *just-enough* rule sends each request to the weakest instance that can still meet its deadline and rejects requests that no instance can serve in time; and periodic risk checks migrate endangered requests by token IDs, 7–15$\times$ cheaper than transferring KV cache. On a four-GPU heterogeneous testbed serving three agentic benchmarks, *GoodServe* attains the highest goodput at every SLO level, up to 31.0% over the best existing router, and its routing overhead stays at 5 ms per request at 10,000 requests per second across 512 simulated instances.

1 INTRODUCTION

An agentic application does not consume an LLM's answer token by token; it waits for the whole answer and then acts on it. A coding agent applies a patch only once the patch is complete Liu et al. (2025), a database agent executes a query only once the query is complete Phan et al. (2025), and a reasoning agent cannot verify a chain of thought before it ends Yan et al. (2025). The latency that matters to such applications is therefore the *end-to-end* latency of an inference, from request to last token, rather than time-to-first-token or time-per-output-token, and users attach explicit end-to-end deadlines (E2E-SLOs) to agentic inferences Zhang et al. (2025); Kim et al. (2026); Yu et al. (2026); Chow (2025); Bari et al. (2025). What matters to the operator is then *goodput*: the number of inferences completed within their deadlines.

Operators serve these inferences on whatever GPUs they own, and what they own is often heterogeneous: GPU generations are purchased over years and pooled together, as in Microsoft's Office 365 Copilot pool of H100, A100, and MI300X GPUs Asgar et al. (2025); Mei et al. (2025). Every incoming request is routed by a proxy to one instance in the pool, and that decision fixes how fast the request will be served: the same inference runs several times slower on a V100 than on an H800, and slower again as the batch on an instance grows (Fig. 1). Routing therefore decides goodput.

Existing routers decide it badly, because they are *greedy*. Random, round-robin, and least-request routing Ray Project (2026b), lowest-TPM routing BerriAI (2026), Llumnix Sun et al. (2024), and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

prefix-cache-aware routers such as Preble Srivatsa et al. (2024) all answer one question per request: which instance serves *this* request fastest, or is least loaded right now. None of them asks whether the request needs to be served fast at all. A request with a loose deadline is still sent to the strongest idle GPU, where it occupies capacity that a later, urgent request will need, while a weaker GPU that would have met the loose deadline stays idle. Goodput requires the opposite discipline, *globally coordinated* routing: place each request only as well as its deadline demands, so that the pool as a whole meets as many deadlines as possible, even if some requests are placed sub-optimally for themselves. The gap between the two is large: on a four-GPU testbed, an oracle router that knows each request's output length and each GPU's speed reaches a goodput of 7.51 requests per second, whereas least-request, round-robin, and Llumnix reach 4.75, 4.82, and 5.04 (Fig. 2), a 49–58% loss that no load balancing recovers.

Coordinated routing is hard for two reasons. First, it must weigh many factors at once, on both the *demand* and the *resource* side. Whether a weaker instance can still meet a deadline depends on the request's input and output lengths, on how much of its prompt that instance already holds in its prefix cache, and on that instance's queuing delay and per-token prefill and decode speed; without the resource side, a router cannot tell a safe demotion from a fatal one, which is exactly why greedy routers, having no such estimate, play safe and send everything to the strongest instance. Second, the most important factor, the output length, is unknown until the request finishes, and instance speeds drift with load, so any estimate will sometimes be wrong; a router that acts on estimates must also be able to correct them.

GoodServe is a goodput-centric router for agentic LLM inferences over heterogeneous GPUs built on these two observations. It formulates routing as a global goodput-maximization problem whose latency model couples demand and resource terms, including prefix-cache reuse (Sec. 3.1), and it solves that problem in a *predict-and-rectify* manner. For demand, a mixture-of-experts predictor exploits the fact that in agentic workloads the output length is governed by an implicit task type: small experts specialize in length tiers and a gate conditioned on the prompt selects among them, at 2.5 ms per request. For resources, an EMA-smoothed black-box estimator learns each instance's queuing delay and per-token speeds from its observed service times, requiring no access to engine internals (Sec. 3.2). For the decision, a *just-enough* rule sends each request to the weakest instance that is still predicted to meet its deadline, which is the per-request realization of the global principle above and costs $O(M)$ over M instances; requests with no feasible instance are rejected, and every τ iterations *GoodServe* re-evaluates the deadline risk of active requests and migrates endangered ones to a stronger instance by transferring their token IDs rather than their KV cache, which is 7.1–$15.3\times$ faster (Sec. 3.3).

We implemented *GoodServe* in 2.5k lines of code and evaluated it on a testbed of four heterogeneous GPUs (V100, A40, A800, H800) serving Llama-3.1-8B and Qwen2.5-14B on a workload mixed from BIRD-bench, SWE-bench, and LiveCodeBench with production arrival traces. *GoodServe* delivers the highest goodput and lowest SLO violation ratio at every SLO level, up to 31.0% over the best existing router, and its advantage grows as deadlines loosen, exactly where coordinated routing has room to act. An oracle output-length predictor adds only 5.1% more goodput, so the routing principle, not prediction accuracy, carries most of the gain; the gain persists under bursty arrivals, out-of-distribution tasks, and a predictor transferred from another model, and the router's overhead stays at 5 ms per request at 10,000 requests per second across 512 simulated instances.

2 Background and Motivation

2.1 Serving Agentic LLM Inferences over Heterogeneous Instances

Confronting booming inference demand, production providers maintain a pool of LLM serving instances, and a service proxy routes every arriving request to one instance for execution. The serving capability of these instances is *inconsistent*, for two reasons. The first is static hardware heterogeneity: providers keep purchasing new GPUs while older ones remain in service, so the pool spans generations Mei et al. (2025); Asgar et al. (2025); Microsoft, for example, serves the Office 365 Copilot workload from a cluster mixing NVIDIA H100, A100, and AMD MI300X GPUs Asgar et al. (2025). The second is dynamic fluctuation: the per-iteration time of an instance depends on the batch it is currently serving, which changes with the instantaneous request intensity. Fig. 1 shows both effects for a fixed sequence across A40, V100, A800, and H800 GPUs and varying batch sizes.

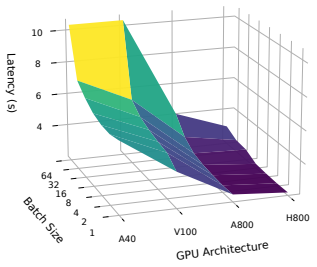


Figure 1: Inference latency across four GPU architectures under varying batch sizes, for a fixed sequence of 100 input and 200 output tokens.

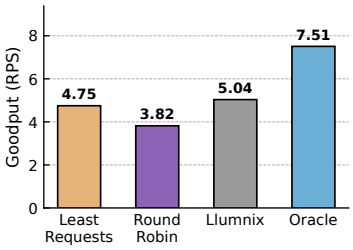


Figure 2: Performance inferiority of existing routing strategies: 600 requests (10 requests per second) served by four heterogeneous GPUs (V100, A40, A800, H800); each request has 100 input tokens and an output length sampled from [100, 500]; the E2E-SLO is 6 s.

For agentic LLM inferences, whose utility is binary upon completion Zhang et al. (2025); Kim et al. (2026) and which carry explicit E2E-SLOs Yu et al. (2026); Zhang et al. (2025); Chow (2025); Bari et al. (2025); Asgar et al. (2025), the routing decision over such a pool determines whether a request meets its deadline, and hence the goodput of the service. Given the heterogeneity above and the uncertainty of decode lengths, making that decision well is both crucial and challenging. We next examine how existing routers make it.

2.2 Lessons Learned from Existing Request-routing Methods

A series of routing methods is used in practice. *Random* (power of two choices) Ray Project (2026a) and *round-robin* Yu et al. (2025) spread requests evenly across instances. *Least-request* and *prefix-cache* routing are built into production frameworks such as AIBrix Team et al. (2025): the former routes to the instance with the fewest pending requests, the latter to the instance with the highest cache hit ratio. *Lowest-TPM* routing in LiteLLM BerriAI (2026) picks the instance with the least tokens-per-minute utilization, the *Preble*-style method Srivatsa et al. (2024); Qin et al. (2025) weighs prefix-cache hits against compute load to find the best-performing instance for a request, and Llumnix Sun et al. (2024) routes to the instance with the most available memory and migrates requests at run time to rebalance load.

All of these are *greedy* and *local*: each request is placed where it, taken alone, would run fastest or where the load is lightest, without regard to its own deadline or to the requests that will arrive after it. They therefore treat urgent and relaxed requests alike, and cannot make a *locally-suboptimal yet globally-optimal* decision, such as sending a relaxed request to a weak instance to keep a strong one free. Fig. 2 quantifies the cost on a four-GPU testbed: an oracle router with ground-truth knowledge of output lengths and instance speeds (policy in Sec. 3.3) reaches a goodput of 7.51, whereas least-request, round-robin, and Llumnix reach 4.75, 4.82, and 5.04.

The lesson is that goodput requires *globally coordinated*, SLO-adaptive routing that reasons about both the demand of each request (its decode length) and the capability of each resource (its speed and queue), rather than a fixed local heuristic. Doing so under the uncertainty of both quantities is the problem this paper addresses.

3 *GoodServe* Design

3.1 Goodput-Optimal Routing as a Global Problem

We first state the routing problem precisely, because the statement itself carries the first contribution of this paper: goodput-optimal routing is a *global* problem over all requests and all instances, and its latency model must couple the *demand* side of a request with the *resource* side of an instance. A detailed symbol table is given in Appendix A.1.

Given the request set $\mathcal{R}$ and the GPU instance set $\mathcal{G}$, let $x_{r,g} \in \{0, 1\}$ indicate whether request r is routed to GPU-g, and let $T(r, g)$ be the end-to-end latency of request-r when served on GPU-g. The

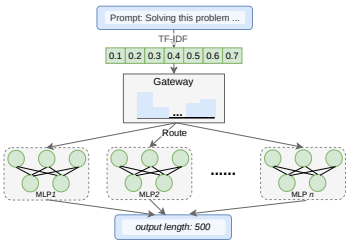


Figure 3: The output-length predictor: TF-IDF features, a gating router, and K MLP experts.

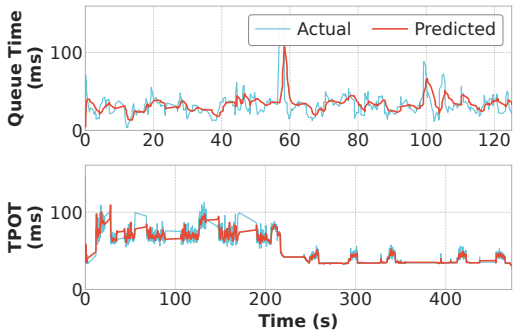


Figure 4: Estimated versus actual queuing time and TPOT on an A40 instance replaying the Mooncake trace.

objective is to maximize goodput,

$$\max_{\{x_{r,g}\}} \sum_{r\in\mathcal{R}} \mathbb{I}\left[\sum_{g\in\mathcal{G}} x_{r,g}\, T(r,g) \le D_r\right], \tag{1}$$

where D_r is the E2E-SLO of request-r, subject to $\sum_g x_{r,g} = 1$ for every request and to the memory capacity of each GPU, which bounds the number of requests it can serve concurrently vLLM Project (2026).

The latency $T(r,g)$ is where demand meets resource. On the demand side, let L_r^{in} and L_r^{out} be the input and output lengths of request-r; on the resource side, let q_g be the average queuing delay on GPU-g and p_g and d_g its average per-token prefill and decode latencies. Prefix caching, which production systems adopt to skip the prefill of cached prefixes Qin et al. (2025), enters through the hit length $H_{r,g}$ of request-r on GPU-g, so that

$$T(r,g) \triangleq q_g + p_g \cdot (L_r^{\text{in}} - H_{r,g}) + d_g \cdot L_r^{\text{out}}. \tag{2}$$

Eq. 2 makes three things explicit that greedy routers leave implicit. Routing is coupled across requests, since q_g and the capacity of GPU-g depend on the requests already placed there. Feasibility on a weaker instance can only be judged with the resource terms, since a large d_g may be tolerable for a short L_r^{out} and fatal for a long one. And cache locality is not a separate objective but a latency term: a long cached prefix $H_{r,g}$ lowers $T(r,g)$ and can make a busier or weaker instance feasible, so *GoodServe* honors locality exactly to the extent that it helps meet the deadline.

Why the problem is hard. Even with exact coefficients, Problem 1 is NP-hard: with binary routing variables and bounded per-GPU capacities, choosing which requests to serve on which GPU so as to satisfy the most deadlines contains the multiple-knapsack problem, where each GPU is a knapsack whose capacity is consumed by the queuing delay and memory that admitted requests add Zhang et al. (2025). The coefficients are, moreover, not exact: L_r^{out} is unknown until the request finishes, and q_g, p_g, d_g drift with the load on each instance. *GoodServe* therefore adopts a *predict-and-rectify* principle. It estimates the demand and resource coefficients accurately enough to make a coordinated decision (Sec. 3.2), decides with a rule that realizes the global principle at $O(M)$ cost per request (Sec. 3.3), and revisits the decision while the request runs so that estimation errors are corrected rather than suffered.

3.2 Practical and Accurate Estimation of Demand and Resources

Both sides of Eq. 2 must be estimated online on the proxy router, which serves thousands of requests per second; both estimators must therefore be *practical*, cheap and free of engine internals, and *accurate enough* for the routing decision, which needs the right feasibility set and the right ordering of instances rather than exact latencies.

Demand: output-length prediction. Agentic inferences span a much wider output-length range than chatbot inferences because they span task types, and the task type is the dominant predictor of

length: a database-operating request Phan et al. (2025) is short, a code-generation request Liu et al. (2025) long. This condition is visible in the prompt but must be handled *implicitly*, without assuming a fixed task set. Existing predictors either pay for a Transformer encoder, as S^3 Jin et al. (2023) does with a fine-tuned DistilBERT, or lose accuracy by ignoring the prompt, as history-based Gong et al. (2025) and single-MLP Wang et al. (2025) predictors do. *GoodServe* instead uses a mixture of small experts (Fig. 3): a feature vector $\mathbf{h}_r$ is obtained by TF-IDF vectorization Sparck Jones (1972) of the token window so far, a two-layer gating router outputs $\mathbf{p}_r = \mathit{softmax}(R(\mathbf{h}_r)) \in \Delta^K$, and the prediction is the weighted combination $\hat{y}_r = \sum_{k=1}^{K} p_{r,k}\, E_k(\mathbf{h}_r)$ of K four-layer MLP experts, 45.1M parameters in total. Training proceeds in two phases: half of the data is partitioned into K subsets by discretizing input and output lengths into $\sqrt{K}$ tiers each ($K = 9$ in practice) and one expert is trained per subset; the router is then trained on the other half with the experts frozen, so that its gate learns to recognize the implicit task type. TF-IDF is chosen for cost, not accuracy: replacing it by learned sentence embeddings with the same architecture changes the normalized error only from 1.00 to 0.98, while the fine-tuned DistilBERT predictor reaches 1.42 (Appendix A.3), and TF-IDF needs no encoder on the router, keeping prediction at 2.5 ms per request. Mixture-of-experts prediction has been used before, by MoPE in Equinox Wei et al. (2025), but for a fundamentally different job: MoPE predicts once, on arrival, to score fairness inside a single server, whereas our predictor exists to drive a global routing decision across heterogeneous instances and to be re-checked while the request runs, so it must be cheap enough for a proxy that handles thousands of requests per second, re-invocable on the generated prefix, and able to signal its own uncertainty.[1]

Resources: instance-capability estimation. The resource coefficients q_g, p_g, and d_g are hard to compute from configuration: under continuous batching in engines such as vLLM vLLM Project (2026), the speed a request experiences depends on the shapes of all requests sharing the batch, and it changes over the many decode steps of an agentic inference. Two observations make estimation practical. First, the routing decision needs a valid ordering of instances and a valid feasibility set, not exact latencies. Second, an instance batches many requests per iteration and its configuration rarely changes, so by the law of large numbers Révész (2014) its per-iteration time is stable over short periods. *GoodServe* therefore estimates q_g, p_g, and d_g in a *black-box* manner, from their recently observed values rather than from engine internals, and smooths them with an exponential moving average against transient jitter. Fig. 4 shows that the estimates track the actual queuing time and TPOT closely when replaying the Mooncake trace Qin et al. (2025) with mixed BIRD-bench Li et al. (2023a) and LiveCodeBench Jain et al. (2024) requests on an A40 GPU.

3.3 Globally Coordinated Routing with Runtime Rectification

Just-enough selection. With the estimates in hand, *GoodServe* realizes the global principle with a rule that is simple to state and cheap to run. Since Fig. 1 shows that stronger GPUs are faster for every request, a greedy router would send every request to the strongest available one; but finishing a request earlier than its deadline adds nothing to goodput and takes capacity from the urgent requests that follow. *GoodServe* therefore computes the feasible set $\mathcal{C}_r = \{g : T(r, g) \le D_r\}$ from Eq. 2 and routes the request to the instance in $\mathcal{C}_r$ with the *weakest* capability, i.e., the largest d_g, reserving stronger instances for requests that need them. This is not an engineering shortcut but the per-request form of the global objective: each request consumes the least capable resource that keeps it within its deadline, which is precisely the locally-suboptimal, globally-optimal behavior that Sec. 2.2 found missing in existing routers, at a cost of $O(M)$ per request in an M-instance cluster. When $\mathcal{C}_r$ is empty, no instance is predicted to meet the deadline and *GoodServe* rejects the request, so that capacity is not spent on work that cannot count toward goodput; Sec. 5.3 shows that this admission rule alone contributes 7.7% goodput. The complete procedure is given in Appendix A.2.

Runtime rectification. Because the coefficients are estimates, a one-shot decision can turn out wrong: the output may run longer than predicted, or the chosen instance may slow down under new load. After every τ inference iterations ($\tau = 50$ by default), *GoodServe* re-predicts the remaining output length of each active request from its generated prefix and refreshes the instance estimates;

[1] Technically, MoPE hard-routes each prompt to one of three fine-tuned BERT experts partitioned by output length; ours softly mixes nine lightweight MLP experts partitioned by joint input–output tiers through a learned gate, is re-invoked every τ iterations on the generated prefix, and exposes the dispersion of expert outputs as a confidence signal.

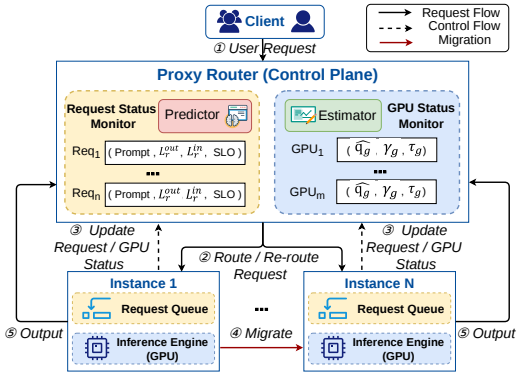


Figure 5: *GoodServe* architecture and workflow.

if the expected finish time now exceeds the deadline, the proxy migrates the request to a stronger instance, again chosen by the just-enough rule. Migration transfers the *token IDs* rather than the KV cache: agentic inferences have long contexts and many decode iterations Wu et al. (2025), so the KV cache is large while recomputing the prefill is cheap, and token-ID transfer trades a small prefill cost for a 7.1–15.3× lower migration latency (Fig. 9), which is what makes rectification affordable at run time.

4 Implementation

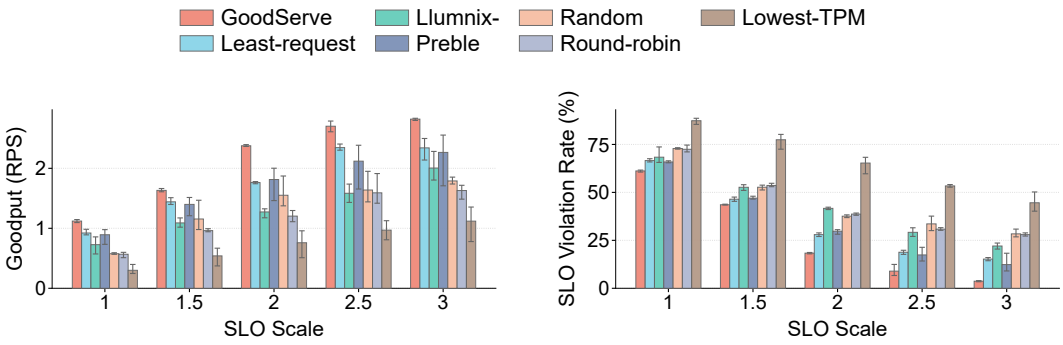
Fig. 5 shows how the pieces fit into a serving stack. A `RequestStatusMonitor` and a `GPUStatusMonitor` on the proxy router maintain the demand- and resource-side estimates of Sec. 3.2; the router applies the just-enough rule of Sec. 3.3 to each arriving request and issues migrations when the periodic risk check fires. We implemented *GoodServe* in about 2.5k lines of Python on top of vLLM vLLM Project (2026); the source code is provided as supplementary material. Each backend runs an unmodified inference engine except for a hook that re-predicts the remaining output length every τ generated tokens, with all such predictions batched on the backend, and reports SLO risk to the proxy through a `/check_migration` interface; the proxy then re-evaluates feasibility, selects the target instance, and resumes generation there from the transferred token IDs. The demand predictor is trained offline on 8,680 samples for 500 epochs, which takes 25 minutes, and is loaded once on the proxy and on each backend. Because prediction is batched and routing is $O(M)$, the per-request overhead of the router stays at milliseconds even for hundreds of instances (Sec. 5.3).

5 Evaluation

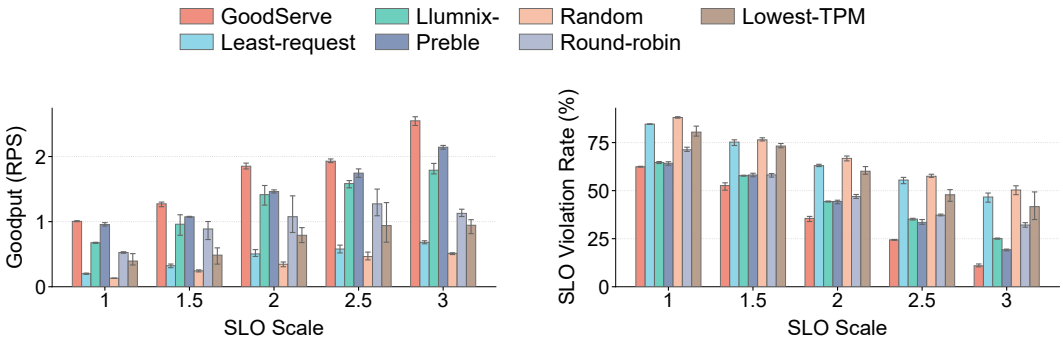
5.1 Experimental Setup

Hardware and models. Our experiments are conducted on a heterogeneous cluster of four instances, each equipped with distinct NVIDIA GPUs: H800 (80GB), A800 (80GB), A40 (48GB), and V100 (32GB, with a TP level of 2). These instances are interconnected by a 10Gbps Ethernet network, and the proxy router is hosted on the A40 instance. Four GPU generations is the scale at which heterogeneous serving is studied in the literature Griggs et al. (2024), and it is the degree of heterogeneity, not the raw GPU count, that routing exploits; routing overhead at cluster scale is measured separately with the real router over simulated backends (Sec. 5.3). We use two representative instruction-tuned models, Llama3.1-8B-Instruct Grattafiori et al. (2024) and Qwen2.5-14B-Instruct Yang et al. (2025), as backend LLMs.

Workloads. We construct a mixed workload suite from three popular agentic benchmarks: BIRD-bench Li et al. (2023a), SWE-bench Jimenez et al. (2023) and LiveCodeBench Jain et al. (2024). We replay the Mooncake production traces Qin et al. (2025) to set the request submission time. To set the E2E-SLOs, following established methodologies Li et al. (2023b); Peng et al. (2025); Zhang et al. (2025), we first measure the median request execution time by running each request alone on a



(a) Performance results on Llama3.1-8B-Instruct



(b) Performance results on Qwen2.5-14B-Instruct

Figure 6: End-to-end performance under different request routing methods.

mid-tier GPU (NVIDIA A800), and then scale that base latency by multiple relaxation factors ($1\times$, $1.5\times$, $2\times$, $2.5\times$ and $3\times$) to emulate diverse SLO urgency levels. The temperature is set to 0 to ensure output consistency across the deadline-setting stage and the testbed-measurement stage. Prefix-cache hits are emulated by hashing prompts and skipping the prefill of cached blocks, which reproduces their effect on latency exactly; on this workload, 29% of prompt tokens (107 of 460 on average) hit a cached prefix on the chosen instance.

Baselines and Metrics. We compare *GoodServe* with diverse baselines described in Sec. 2.2: the built-in *random* and *least-request* methods in AIBrix Team et al. (2025), as well as the *round-robin* Yu et al. (2025), *lowest-TPM* BerriAI (2026), *Preble* Srivatsa et al. (2024) and *Llumnix* Sun et al. (2024) methods. Meanwhile, we adopt two metrics: (1) *goodput*, which is the average number of requests completing within their E2E-SLO *per second*, and (2) *SLO violation ratio*, which is the overall ratio of requests failing to satisfy the E2E-SLO requirements.

5.2 End-to-End Performance

In Fig. 6, we depict the end-to-end results across different SLO scales for both 8B and 14B models. We repeat each experiment five times and report the max/min values as error bars. As shown in Fig. 6, *GoodServe* consistently delivers the best goodput and the lowest SLO violation rate. For example, for the 8B model at a medium SLO scale of 2, *GoodServe* surpasses the second-best (*Preble*) by 31.0% in goodput. Meanwhile, *GoodServe* generally provides larger benefits under looser SLO requirements. For the 14B model, its goodput improvement over the second-best baseline ranges from 5.2% to 26.6% across different SLO scales. This is because a looser SLO requirement allows for more *locally-suboptimal* yet *globally-optimal* routing decisions, which existing SLO-unaware methods all fail to make.

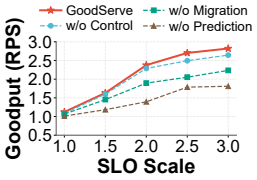
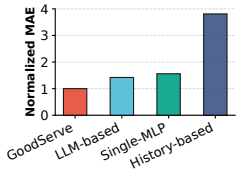
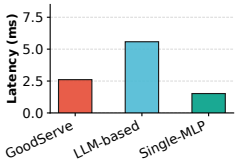


Figure 7: Effectiveness of admission control, prediction, and migration innovations in *GoodServe*.



(a) Prediction error.



(b) Prediction latency.

Figure 8: Accuracy and overhead comparison among different prediction methods.

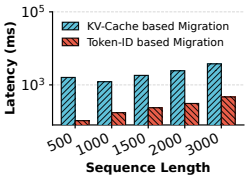
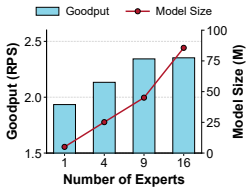
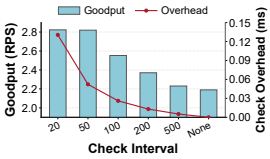


Figure 9: Average migration latency under different state transferring methods.



(a) Number of experts.



(b) Status recheck interval.

Figure 10: Hyper-parameter sensitivity analysis.

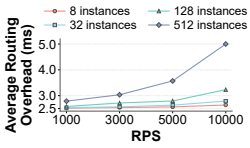


Figure 11: Routing overheads at varying cluster size and request intensity.

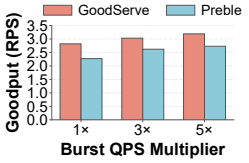


Figure 12: Goodput under varying burst traffic intensity.

5.3 Deep Dive Experiments

In this part, we conduct a series of deep-dive experiments. Unless otherwise specified, we use the 8B model at an SLO scale of 3.

Ablation study. Recall that *GoodServe* adopts admission control, an MoE-style request length predictor, and runtime migration to improve goodput and robustness. To evaluate their necessity, we create three variants: (1) *GoodServe without admission control*, which admits all incoming requests; (2) *GoodServe without prediction*, which replaces the original predictor with a history-based one Gong et al. (2025), as prediction itself cannot be disabled; and (3) *GoodServe without migration*, which disables runtime request migration. As shown in Fig. 7, all three components are indispensable. For example, at an SLO scale of 2.5, removing the admission control, runtime migration, and MoE-style predictor reduces the goodput by 7.7%, 24.0%, and 33.9%, respectively. We note that their importance increases under looser SLOs, because a looser SLO allows for more flexible routing optimizations.

Superiority of our MoE-style demand predictor. We check the superiority of our MoE-style predictor over the other ones discussed in Sec. 3.2, in terms of both accuracy and overhead. Fig. 8 shows the normalized *Mean Absolute Error* (MAE) for different predictors. It suggests that our MoE-style predictor adopted in *GoodServe* can yield the best accuracy—with an error reduction of $1.4\times$ compared to the LLM-based one Jin et al. (2023), and up to $3.8\times$ compared to the history-based one Gong et al. (2025). Meanwhile, its prediction latency (around 2.5 ms per request) is lower than the LLM-based one. Furthermore, replacing our predictor with the LLM-based, Single-MLP, and history-based ones reduces the goodput by 12.5%, 16.7%, and 20.4%, respectively. Moreover, even an Oracle predictor with exact output-length knowledge improves goodput by only 5.1% over our predictor.

Superiority of our Token-ID based request migration method. Recall that in Sec. 3.3, for request migration we choose to transfer the Token-IDs instead of the KV cache. To evaluate its superiority, we refer to Fig. 9, which shows the average migration latency for requests of different lengths. Compared with the KV-cache-based method, token-ID based migration achieves $7.1\times$ to $15.3\times$ lower latency.

Hyper-parameter sensitivity analysis. *GoodServe* introduces two hyper-parameters: the number of experts K in the predictor (Sec. 3.2), and the status checking frequency τ (Sec. 3.3). Here we evaluate *GoodServe*'s performance by varying their values. As shown in Fig. 10a, while adopting more experts can yield higher accuracy and consequently higher goodput, $K = 9$ can already yield sufficiently good performance, much better than $K = 4$ yet comparable to $K = 16$. Meanwhile, regarding τ, Fig. 10b suggests that a higher checking frequency does improve goodput.

Overhead analysis with large-scale simulation. In production clusters, there may be hundreds of instances. Since our core innovation lies in the proxy router instead of on the serving engine, to check *GoodServe*'s scalability we resort to large-scale simulations. Specifically, we configure a set of virtual IPs each corresponding to a simulated local inference engine. We respectively simulate 8, 32, 128 and 512 instances, and for each case we vary the RPS from 1000 to 10000—all requests handled by a single router. As shown in Fig. 11, thanks to the lightweight prediction model (Sec. 3.2), the simple heuristic (Sec. 3.3) as well as the implementation optimizations (distributed and batched predictor execution), the routing overhead of *GoodServe* is indeed quite small: the per-request routing latency is still marginal (5.0 ms) even when serving highly-intense workloads (RPS=10000) over 512 nodes.

Bursty traffic. To evaluate scheduling under bursty arrivals, we inject a controlled local burst into the original trace by selecting a contiguous request segment and compressing its inter-arrival times to increase the local arrival rate to $1\times$, $3\times$, and $5\times$ the original rate, while keeping the remaining trace unchanged. As shown in Figure 12, GoodServe consistently outperforms Preble across all burst intensities, achieving at least 15.6% higher goodput. Appendix A.3 further reports *GoodServe*'s goodput under out-of-distribution tasks, non-zero decoding temperature, a predictor transferred from a different model, an SLO-aware routing baseline distilled from JITServe, and semantic-embedding features for the predictor.

6 Related Work and Discussion

Positioning against heterogeneous and SLO-aware serving systems. Several systems touch heterogeneous or SLO-aware serving, but each decides a different variable. Helix Mei et al. (2025) decides how one model's layers are placed across heterogeneous nodes and which node path a request traverses, targeting aggregate throughput; Mélange Griggs et al. (2024), MILP-based provisioning studies Jiang et al. (2025), SageServe Jaiswal et al. (2025), and *llm-d* llm-d Project (2026) decide the GPU composition, deployment configuration, or scale of a deployment; JITServe Zhang et al. (2025) schedules requests *within* an instance under imprecise request information and does not route across heterogeneous backends; Llumnix Sun et al. (2024) and Preble Srivatsa et al. (2024) route across instances but without deadline awareness, and Llumnix migrates KV cache rather than token IDs. *GoodServe* takes the instance set as given and decides, per request and per deadline, which self-contained instance serves it; it can run standalone or as the routing layer beneath a provisioning system, and it is, to our knowledge, the first routing work for agentic inferences whose primary metric is TTLT. An SLO-aware routing baseline distilled from JITServe's earliest-available-slot principle reaches 1.79 goodput versus 2.68 for *GoodServe* in Appendix A.3, because it remains greedy in its instance choice.

Applicability. For inferences without E2E-SLOs, *GoodServe* can fall back to conventional routing without affecting its other components, and routing relaxed agentic inferences to weaker instances in fact improves the TTFT and TPOT of co-served chatbot inferences. In homogeneous clusters the instances still differ in queuing time and batch size at any instant, so the same rule applies. For PD-disaggregated serving Qin et al. (2025), Eq. 2 can model the prefill and decode terms on the prefill-only and decode-only instances of each candidate pair, and the just-enough rule carries over.

7 Conclusion

Agentic LLM inferences are judged by whether their whole output arrives before a deadline, and serving them on heterogeneous GPUs turns routing into a global goodput problem that greedy, locally optimal routers cannot solve. *GoodServe* formulates that problem with a latency model coupling request demand and instance capability, and solves it in a predict-and-rectify manner: practical estimators for both sides, a just-enough rule that realizes globally coordinated routing at $O(M)$ per request, and token-ID migration that corrects estimation errors at run time. On a heterogeneous testbed it raises goodput by up to 31.0% over the best existing router, and its routing overhead remains negligible even at the scale of hundreds of instances.

AI USE STATEMENT

Generative AI tools were used to polish and restructure author-written text for readability and to assist with the implementation of evaluation scripts; they were not used for research ideation, system design, experiment design, data analysis, or result interpretation. All AI-assisted text and code were reviewed and verified by the authors, who take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

Sec. 5.1 specifies the testbed, models, workloads, SLO construction, and baselines; Sec. 3.2 and Sec. 3.3 give the predictor architecture and training procedure, the estimator update rule, and the routing and migration rules, with the routing algorithm in Appendix A.2; Appendix A.3 reports the robustness experiments. An anonymized implementation of *GoodServe*, the trained predictors, and the run configurations are provided as supplementary material.

REFERENCES

Zain Asgar, Michelle Nguyen, and Sachin Katti. Efficient and scalable agentic ai with heterogeneous systems. *arXiv preprint arXiv:2507.19635*, 2025.

Agrim Bari, Parikshit Hegde, and Gustavo de Veciana. Optimal scheduling algorithms for llm inference: Theory and practice. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 9(3):1–43, 2025.

BerriAI. LiteLLM: Python sdk and proxy server for unified llm api access. `https://github.com/BerriAI/litellm`, 2026. GitHub repository. Accessed: 2026-04-14.

Will Chow. Slice: Slo-driven scheduling for llm inference on edge computing devices. *arXiv preprint arXiv:2510.18544*, 2025.

Ruihao Gong, Shihao Bai, Siyu Wu, Yunqian Fan, Zaijun Wang, Xiuhong Li, Hailong Yang, and Xianglong Liu. Past-future scheduler for llm serving under sla guarantees. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 798–813, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. M\'elange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Shashwat Jaiswal, Kunal Jain, Yogesh Simmhan, Anjaly Parayil, Ankur Mallick, Rujia Wang, Renee St Amant, Chetan Bansal, Victor Rühle, Anoop Kulkarni, et al. Sageserve: Optimizing llm serving on cloud data centers with forecast aware auto-scaling. *arXiv preprint arXiv:2502.14617*, 2025.

Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Guoliang He, Xupeng Miao, Ana Klimovic, Bin Cui, Binhang Yuan, and Eiko Yoneki. Demystifying cost-efficiency in llm serving over heterogeneous gpus. *arXiv preprint arXiv:2502.00722*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. S^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

Hyungjun Kim et al. Kairos: Power-aware serving of agentic ai workloads. *arXiv preprint arXiv:2604.16682*, 2026.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023a.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, 2023b.

Siwei Liu, Jinyuan Fang, Han Zhou, Yingxu Wang, and Zaiqiao Meng. Sew: Self-evolving agentic workflows for automated code generation. *arXiv preprint arXiv:2505.18646*, 2025.

llm-d Project. Workload variant autoscaler. `https://llm-d.ai/docs/architecture/Components/workload-variant-autoscaler`, 2026. Accessed: 2026-05-05.

Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 586–602, 2025.

You Peng, Youhe Jiang, Wenqi Jiang, Chen Wang, and Binhang Yuan. Hexgen-flow: Optimizing llm inference request scheduling for agentic text-to-sql. *arXiv preprint arXiv:2505.05286*, 2025.

Xuan-Quang Phan, Tan-Ha Mai, Thai-Duy Dinh, Minh-Thuan Nguyen, and Lam-Son Lê. Askdb: An llm agent for natural language interaction with relational databases. *arXiv preprint arXiv:2511.16131*, 2025.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170, 2025.

Ray Project. Ray serve documentation. `https://docs.ray.io/en/latest/serve/index.html`, 2026a. Accessed: 2026-01-25.

Ray Project. Ray serve llm routing policies. `https://docs.ray.io/en/latest/serve/llm/architecture/routing-policies.html`, 2026b. Accessed: 2026-01-25.

Pál Révész. *The laws of large numbers*, volume 4. Academic Press, 2014.

Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for llm serving. *arXiv preprint arXiv:2407.00023*, 2024.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pp. 173–191, 2024.

The AIBrix Team, Jiaxin Shan, Varun Gupta, Le Xu, Haiyang Shi, Jingyuan Zhang, Ning Wang, Linhui Xu, Rong Kang, Tongping Liu, et al. Aibrix: Towards scalable, cost-effective large language model inference infrastructure. *arXiv preprint arXiv:2504.03648*, 2025.

vLLM Project. vllm: A high-throughput and memory-efficient inference and serving engine for llms. `https://github.com/vllm-project/vllm`, 2026. Accessed: 2026-01-25.

Zhibin Wang, Zetao Hong, Xue Li, Zibo Wang, Shipeng Li, Qingkai Meng, Qing Wang, Chengying Huan, Rong Gu, Sheng Zhong, et al. Adaptive rescheduling in prefill-decode disaggregated llm inference. *arXiv preprint arXiv:2510.13668*, 2025.

Zhixiang Wei, James Yen, Jingyi Chen, Ziyang Zhang, Zhibai Huang, Chen Chen, Xingzi Yu, Yicheng Gu, Chenggang Wu, Yun Wang, Mingyuan Xia, Jie Wu, Hao Wang, and Zhengwei Qi. Equinox: Holistic fair scheduling in serving large language models. arXiv preprint arXiv:2508.16646, 2025.

Haoran Wu, Can Xiao, Jiayi Nie, Xuan Guo, Binglei Lou, Jeffrey TH Wong, Zhiwen Mo, Cheng Zhang, Przemyslaw Forys, Chengyang Ai, et al. Combating the memory walls: Optimization pathways for long-context agentic llm inference. *arXiv preprint arXiv:2509.09505*, 2025.

Yibo Yan, Shen Wang, Jiahao Huo, Philip S Yu, Xuming Hu, and Qingsong Wen. Mathagent: Leveraging a mixture-of-math-agent framework for real-world multimodal mathematical error detection. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pp. 69–82, 2025.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Jiahuan Yu, Mingtao Hu, Zichao Lin, and Minjia Zhang. Superinfer: Slo-aware rotary scheduling and memory management for llm inference on superchips. *arXiv preprint arXiv:2601.20309*, 2026.

Shibo Yu, Mohammad Goudarzi, and Adel Nadjaran Toosi. Efficient routing of inference requests across llm instances in cloud-edge computing. *arXiv preprint arXiv:2507.15553*, 2025.

Wei Zhang, Zhiyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. Jitserve: Slo-aware llm serving with imprecise request information. *arXiv preprint arXiv:2504.20068*, 2025.

A Appendix

A.1 Notations used in *GoodServe*

Notation	Description
r, g	Request index, GPU instance index
$\mathcal{R}$, $\mathcal{G}$	Set of requests, set of GPU instances
D_r	End-to-end latency deadline (E2E-SLO) of request r
L_r^{in}, L_r^{out}	Input length and (predicted) output length of request r
$H_{r,g}$	Reusable (cache-hit) prefix length of request r on GPU g
q_g	Estimated average queuing delay on GPU g
p_g, d_g	Estimated average per-token prefill and decode latencies on GPU g
$x_{r,g}$	Binary routing variable: 1 if request r is routed to GPU g
$T(r,g)$	Predicted end-to-end latency of request r if routed to GPU g (Eq. 2)
$\mathcal{C}_r$	Feasible instance set of request r, $\{g : T(r,g) \le D_r\}$
τ	Status re-check interval (inference iterations)
$\mathbb{I}[\cdot]$	Indicator function

Table 1: Key notations used in the routing formulation of *GoodServe*.

A.2 Routing Algorithm

Algorithm 1 Just-enough instance selection for an arriving request r

$\mathbf{h}_r \leftarrow \textsc{BuildFeatures}(r)$; $L_r^{\text{out}} \leftarrow \mathcal{P}(\mathbf{h}_r)$ ▷ demand prediction
$\mathcal{C}_r \leftarrow \emptyset$
for all $g \in \mathcal{G}$ **do**
  $H_{r,g} \leftarrow \textsc{ReusePrefix}(r,g)$; $(q_g, p_g, d_g) \leftarrow \textsc{InstanceEstimate}(g)$ ▷ resource estimation
  $T(r,g) \leftarrow q_g + p_g\,(L_r^{\text{in}} - H_{r,g}) + d_g\,L_r^{\text{out}}$
  if $T(r,g) \le D_r$ **and** GPU-g has memory for r **then**
    add g to $\mathcal{C}_r$
  end if
end for
if $\mathcal{C}_r = \emptyset$ **then**
  return $\textsc{Reject}(r)$
end if
return $g^\star \leftarrow \arg\max_{g \in \mathcal{C}_r} d_g$ ▷ weakest feasible instance

Algorithm 1 is invoked once when a request arrives and again whenever the periodic risk check of Sec. 3.3 finds that an active request is no longer expected to meet its deadline on its current instance; in the latter case the request's remaining output length replaces L_r^{out}, its current instance is excluded from $\mathcal{G}$, and the selected instance becomes the migration target.

A.3 Additional Robustness Experiments

Unless noted otherwise, the experiments below use Llama-3.1-8B-Instruct at an SLO scale of 3 on the testbed of Sec. 5.1, and "best baseline" denotes the best-performing baseline of Sec. 5.2 in each setting.

Prediction accuracy versus the routing rule. With the just-enough rule held fixed, we replace the predictor by an oracle that knows the true output length and by the history-based baseline predictor Gong et al. (2025).

Configuration (routing identical)	Goodput (RPS)
Just-enough routing + oracle output length	2.86
Just-enough routing + *GoodServe* predictor	2.68
Just-enough routing + history-based predictor	2.23

Perfect prediction adds 6.7% over our predictor, whereas the baseline predictor loses 16.8%; the goodput that survives even the weakest predictor is attributable to the routing rule itself, and our predictor closes most of the remaining gap.

An SLO-aware routing baseline. JITServe Zhang et al. (2025) schedules within an instance and exposes no inter-instance routing policy. We distill the routing analogue of its principle, earliest-available-slot-first, and compare it end to end; this construction is ours and should be read as a comparison against the principle rather than the JITServe system. It reaches a goodput of 1.79 versus 2.68 for *GoodServe*: earliest-available-slot remains greedy in its instance choice and still occupies the most capable instances with requests that weaker instances could have completed in time.

Out-of-distribution tasks, decoding temperature, and predictor transfer.

Setting	*GoodServe*	Best baseline
Default workload	2.68	2.27
Workload extended with GSM8K requests (one third of the original size)	2.63	2.25
Decoding temperature 0.7	2.67	2.38
Predictor trained for Qwen2.5-14B, serving Llama-3.1-8B	2.59	—

The advantage holds under task drift, sampling-based decoding, and a predictor transferred from a different model, the harshest of the three shifts; runtime rectification absorbs the added prediction error.

Semantic features for the predictor. Replacing TF-IDF features by LaBSE sentence embeddings, with the predictor architecture unchanged, gives a normalized MAE of 0.98 versus 1.00, while the fine-tuned DistilBERT predictor S^3 Jin et al. (2023) gives 1.42. Learned embeddings add no accuracy for this task, where output length is governed by lexical and structural cues of the task type, and they require a 3 GB encoder on the router; *GoodServe* uses TF-IDF for cost, not for accuracy.

Migrations under bursty arrivals. Under the $5\times$ burst intensity of Fig. 12, the number of migrations rises to $1.15\times$ that under normal traffic, which is the rectification mechanism absorbing the burst; the relative advantage of *GoodServe* stays above 15% at every burst level.