Innovation and Technology Studies
Print ISSN: 3007-6919
Online ISSN: 3007-6927
DOI: https://doi.org/10.61784/its3024

PACO: PREDICTIVE AUTO-CONFIGURATION FOR SLO-CONSTRAINED LARGE LANGUAGE MODEL INFERENCE SERVING

Dai Teng
University of Illinois Urbana-Champaign, Illinois, USA.

Abstract: Serving large language models (LLMs) is expensive, and production workloads fluctuate by an order of magnitude within hours while requests must meet tight latency service-level objectives (SLOs) on time-to-first-token (TTFT) and time-per-output-token (TPOT). Operators today either provision statically for peak load, wasting most of the fleet off-peak, or rely on reactive autoscalers that reconfigure only after violations are observed. This paper presents PACO, a predictive auto-configuration framework that periodically selects the cheapest serving configuration expected to satisfy tail-latency SLOs. PACO combines (i) a lightweight workload forecaster, (ii) a gradient-boosted performance model that predicts windowed P95 TTFT/TPOT from workload features and a candidate configuration, learned from offline profiling, (iii) an SLO-constrained minimum-cost search over the joint space of replica count and chunked-prefill token budget, and (iv) a feedback guard that bounds the impact of prediction errors. We evaluate PACO with an iteration-level cluster simulator driven by production LLM inference traces released by Microsoft Azure. On a diurnal conversation workload, PACO reduces SLO-violating windows from 27.6% to 3.4% at 2% lower cost than a reactive autoscaler, cuts cost by 60.5% relative to static peak provisioning, and stays within 27% of a clairvoyant oracle's cost. Ablations show the forecaster halves violations and the joint tuning of the token budget is essential for long-context workloads.
Keywords: Large language models; Inference serving; Auto-configuration; Autoscaling; Service-level objectives; Performance prediction

1 INTRODUCTION

Transformer-based large language models (LLMs) now power interactive services such as chat assistants and coding copilots [1,2]. Serving them is dominated by expensive, power-hungry GPU fleets, and providers face a hard dual objective: each request must meet latency service-level objectives (SLOs)—typically on the time-to-first-token (TTFT) and the time-per-output-token (TPOT)—while the cost of the provisioned fleet must be minimized. Production traces show that request rates and token-length mixes fluctuate substantially over hours and can spike within minutes [3,4], so no single deployment configuration is simultaneously cheap and safe.

The configuration space of a modern LLM serving engine is also richer than classical model serving. Beyond the number of model replicas, engines built around continuous batching and chunked prefill expose intra-engine knobs—most prominently the per-iteration token budget—that trade TTFT against TPOT and interact non-linearly with load and prompt-length distributions [5,6]. Choosing a configuration therefore requires answering a predictive question: which configuration will satisfy the tail-latency SLOs of the workload that is about to arrive, at the lowest cost?

Operators today answer it poorly. Static peak provisioning meets SLOs but, in our trace-driven experiments, costs 3.2× more than a clairvoyant per-window optimum. Reactive autoscalers in the style of the Kubernetes Horizontal Pod Autoscaler reconfigure only after violations are measured; on a diurnal workload derived from Azure production traces, a well-tuned reactive policy still violates SLOs in 27.6% of one-minute control windows because it perpetually lags load ramps. Bayesian-optimization-style configuration search finds good static configurations offline but cannot follow dynamic load [7,8], while recent LLM-specific systems reconfigure for energy or split phases across machine pools using expert-built analytical models rather than a learned, workload-conditioned latency predictor [3,4].

This paper presents PACO (Predictive Auto-COnfiguration), a control framework that closes this gap. Every control window, PACO (i) forecasts the next window's request rate with a Holt linear-trend smoother and tracks token-length statistics, (ii) queries a learned performance model—gradient-boosted trees trained on offline profiling data—that maps (workload features, configuration) to windowed P95 TTFT and P95 TPOT, (iii) searches the joint space of replica count and chunked-prefill token budget for the cheapest configuration whose predicted tail latencies clear the SLOs with a safety margin, and (iv) applies a reactive guard that bounds the damage of forecast or model errors.

We make the following contributions. First, we formulate SLA-constrained auto-configuration of LLM serving as a per-window constrained cost-minimization problem over both fleet-level and engine-level knobs, and present a practical predict-then-optimize controller for it. Second, we build an iteration-level discrete-event simulator of a multi-replica continuous-batching cluster with chunked prefill and KV-cache limits, calibrated to published A100-class measurements, and drive it with real production arrival and token-length traces from Microsoft Azure [3]. Third, we show end to end that PACO reduces SLO-violating windows from 27.6% to 3.4% at slightly lower cost than a reactive autoscaler, cuts cost by 60.5% versus static peak provisioning, and remains within 27% of a clairvoyant oracle. Fourth, ablations and a long-

context code workload yield two systems insights: the workload forecaster halves violations relative to using last-window observations, and jointly tuning the token budget is essential—for long prompts the oracle nearly always prefers a small chunked-prefill budget to protect TPOT—while statistically unpredictable arrival spikes bound what any causal policy can achieve.

2 BACKGROUND AND MOTIVATION

2.1 LLM Inference Serving and Its SLOs

An LLM request is processed in two phases: a compute-bound prefill over the prompt, which produces the first output token, and a memory-bound autoregressive decode that emits the remaining tokens [3]. Production engines interleave both phases across requests with continuous (iteration-level) batching [5], keep per-request key–value (KV) caches in GPU memory [9], and bound each scheduling iteration by a token budget so that long prefills are chunked and co-scheduled with ongoing decodes [6]. Interactive services attach SLOs to the two user-visible latencies: TTFT, the delay until the first token, and TPOT, the average inter-token delay thereafter. Because means hide queueing spikes, SLOs are stated on tail percentiles; we use windowed P95 targets throughout.

2.2 The Configuration Problem

We consider two knobs that an operator can change online without touching the model: the number of replicas R (each occupying one GPU), which sets both capacity and cost, and the per-iteration token budget B, which governs the throughput–latency trade-off inside each replica: a large budget accelerates prefill (better TTFT under load) but lets prefill chunks inflate the iterations in which decodes participate (worse TPOT), and vice versa [6]. The two knobs interact with the workload. Our experiments in Sec. V-D show that for a conversation workload the oracle prefers B = 2048 in 57 of 58 windows, whereas for a long-prompt code workload it selects B = 512 almost always—replica count alone cannot express this. The action space in our evaluation is $R \in \{1, \ldots, 14\} \times B \in \{512, 2048, 4096\}$.

2.3 Why Reactive Scaling Is Not Enough

A reactive controller measures the last window and corrects afterwards, so every sustained ramp begins with a violation, and the correction (typically multiplicative increase) overshoots, after which scale-down undershoots again. In our evaluation the reactive baseline violated 16 of 58 windows, almost all during the two load ramps. A predictive controller can instead provision for the ramp: with a one-window-ahead rate forecast whose mean absolute percentage error (MAPE) is 15%, the required capacity is known before the load arrives. Prediction alone, however, is unsafe—flash crowds are not forecastable from history—which motivates PACO's hybrid predict-then-guard design.

3 PACO DESIGN

PACO operates on a fixed control interval (60 s in our evaluation). At each window boundary it executes four stages and outputs a configuration (R, B) for the next window. Scale-ups add empty replicas immediately; scale-downs drain removed replicas, which finish their resident requests without admitting new ones and are billed until empty.

3.1 Workload Monitoring and Forecasting

The monitor aggregates, per completed window, the request rate and the token-length statistics of arrivals (mean and P95/P99 of prompt length, fraction of prompts longer than 4096 tokens, mean and P95 of output length). The request rate—the fastest-moving signal—is forecast one window ahead with Holt's linear-trend exponential smoothing ($\alpha = 0.5$, $\beta = 0.3$) [10], clipped to $[0.3, 2.5]\times$ the current level for robustness on sparse workloads. Because under-provisioning is far costlier than over-provisioning, PACO uses an upward-biased estimate, the maximum of the Holt forecast and the last observed rate: on down-ramps it descales one window late, on up-ramps it follows the trend. Token statistics, which drift slowly, are tracked with an exponentially weighted moving average (weight 0.5).

3.2 Learned Performance Model

The core of PACO is a model Ł(w, c) that predicts the windowed P95 TTFT and P95 TPOT that configuration c = (R, B) would deliver under workload features w. We train one gradient-boosted regression-tree ensemble per target on the logarithm of the latency (histogram-based implementation [11,12]; 400 trees, depth ≤ 6, learning rate 0.06). The training corpus is produced by offline profiling: 255 workload windows are sampled from the two Azure traces with arrival rates rescaled by random factors in [0.25, 6], and each window is replayed against 12 random configurations from the action space, yielding 3060 (w, c) → latency examples. Windows that saturate (requests unfinished long after the window) are censored at 60 s so the model learns the location of the capacity cliff. Profiling is a one-time cost per hardware/model pair and is embarrassingly parallel; the same simulate-to-profile methodology is used by recent capacity-planning tools [13].

3.3 SLO-Constrained Minimum-Cost Search

Given forecast features $\hat{w}$, PACO selects the configuration that minimizes cost subject to the predicted SLOs: it scans R in increasing order (cost is proportional to R) and, at the first R for which any budget B is predicted feasible, picks among feasible budgets the one with the largest normalized SLO headroom, a tie-break that is robust to residual model error. Feasibility requires predicted P95 TTFT $\leq 0.8 \cdot$SLO and predicted P95 TPOT $\leq 0.7 \cdot$SLO; the asymmetric margins reflect the model's slightly higher TPOT error (Sec. V-A). If no configuration is predicted feasible, the maximum configuration is deployed. With 42 candidate configurations the whole decision, including model inference, takes about 0.1 s on a CPU core—negligible against the 60 s control interval.

3.4 Reactive Guard

Prediction fails when arrivals are statistically unpredictable. PACO therefore monitors an online violation signal for the window just finished: the measured P95s exceeded the SLOs, or more than 30% of the window's arrivals were still unfinished at the boundary (a backlog indicator that is robust to requests legitimately in flight). When the signal fires, the deployed replica count is raised to at least the current count plus two, overriding the predictive choice. The guard converts a potential run of violating windows into at most a short one and never engages in steady state.

4 EXPERIMENTAL METHODOLOGY

4.1 Trace-Driven Cluster Simulator

Evaluating closed-loop auto-configuration on a physical cluster would require re-running hour-long workloads for every policy and sweeping thousands of profiling configurations, which is impractical; like recent work on LLM serving configuration [4,13], we therefore use high-fidelity simulation. We built an iteration-level discrete-event simulator of a multi-replica cluster with continuous batching, Sarathi-style chunked prefill under a token budget [6], FCFS admission from a shared queue, a per-replica KV-cache capacity of 160 k tokens with conservative reservation, and at most 256 resident sequences per replica. The duration of one scheduling iteration on a replica is modeled as $t = c_0 + c_1 \cdot$(prefill tokens in the batch) $+ c_2 \cdot$(decoding sequences in the batch), with $c_0 = 5$ ms, $c_1 = 0.125$ ms/token, and $c_2 = 0.4$ ms/sequence—constants consistent with published A100-80GB measurements for 13B-parameter dense models (≈8 k prefill tokens/s; 25–90 ms decode steps depending on batch) [3,6,14]. First tokens are emitted with the final prefill chunk. Reconfiguration follows the drain semantics of Sec. III; cost is accounted as provisioned GPU-seconds, including draining replicas. The simulator is resumable, so controllers observe metrics and reconfigure in a true closed loop.

4.2 Workloads and SLOs

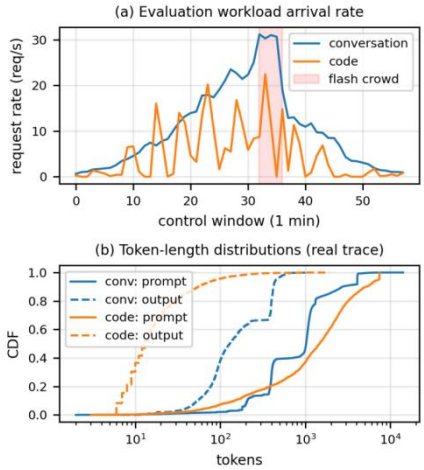

Figure 1 Evaluation Workloads (a) Request Rate Per Control Window after Diurnal Modulation of the Real Azure traces; The Shaded Band is an Injected Flash Crowd. (b) Prompt and Output Token-length CDFs of The Unmodified Production Traces [3]

Arrival processes and token lengths come from the public Azure LLM inference traces released with Splitwise [3]: a conversation service (19,366 requests over ≈1 h; mean prompt 1155 tokens, mean output 211) and a code-completion service (8,819 requests; mean prompt 2048 tokens, P95 7303, mean output 28). Because each raw trace spans only one hour of roughly stationary load, we impose a diurnal-style modulation on the per-minute arrival rate—a sinusoid spanning 0.25–3.0× the native rate plus a two-window flash-crowd step of +2.0× (rate capped at 4.4×)—while drawing inter-arrival times and (prompt, output) pairs from the corresponding real trace window by bootstrap, preserving its burstiness and length correlations. Figure 1 shows the resulting workloads. SLOs are per service class, at P95 within each 60 s window: (TTFT $\leq$ 1.0 s, TPOT $\leq$ 60 ms) for conversation and (TTFT $\leq$ 2.0 s, TPOT $\leq$ 100 ms) for code, whose 7 k-token prompts make sub-second first tokens physically unattainable.

4.3 Policies Under Comparison

We compare: PACO as described; PACO−forecaster, which replaces the rate forecast by the last observed rate; PACO−guard, which disables the reactive override; Reactive, an HPA-style SLO-feedback autoscaler that multiplies R by 1.5 after a violating window and decrements R when both measured P95s are below 75% of their SLOs (B fixed at 2048); Static-Peak and Static-Mean, fixed provisioning at the maximum and the mean of the oracle's per-window replica counts; and Oracle, which selects for every window, with full knowledge of its actual requests, the cheapest configuration that meets the SLOs in an isolated replay—a clairvoyant lower bound on cost. All policies run in the same closed-loop simulation; metrics are the fraction of SLO-violating windows (a window violates if either windowed P95 exceeds its SLO or any of its requests never completes) and total provisioned GPU-hours.

5 EVALUATION

5.1 Performance-Model Accuracy

We hold out all profiling windows drawn from a contiguous 10-minute band of the traces (564 examples) and train on the remainder. On the holdout, the model attains a log-space R^2 of 0.961 for P95 TTFT and 0.897 for P95 TPOT (MAPE 23.9% and 30.7%); most residual error sits near the capacity cliff, where latency is steepest in load. What the controller actually consumes is the feasibility decision, and classification accuracy against the SLO thresholds is 91.8% (TTFT) and 91.5% (TPOT); the safety margins of Sec. III-C absorb most remaining misclassifications. Figure 2 visualizes predictions against measurements.

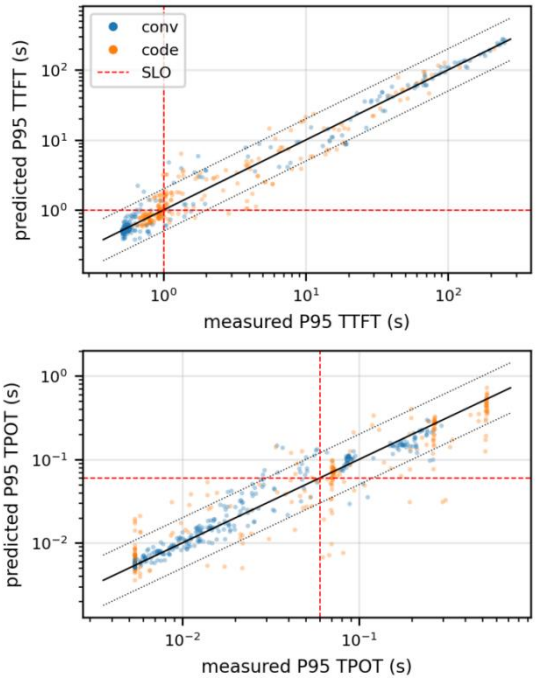

Figure 2 Learned Performance Model on Held-out Profiling Windows: Predicted vs. Measured Windowed P95 TTFT (Top) and P95 TPOT (Bottom). Dotted Lines Mark 2× Error; Dashed Red Lines Mark The Conversation-class SLOs

5.2 End-to-End Results on the Conversation Workload

Table 1 End-to-end Results, Conversation Workload (58 Windows)

Policy	Cost (GPU·h)	Viol. windows	Mean replicas
Oracle	3.41	1.7%	3.26
PACO (ours)	4.32	3.4%	4.21
Reactive	4.42	27.6%	4.29
Static-Peak	10.95	0.0%	9.00
Static-Mean	3.65	69.0%	3.00

Table 1 summarizes the diurnal conversation experiment and Figure 3 shows the control timeline. PACO's biased Holt forecast tracks the rate closely (MAPE 15%, essentially zero lag on the ramps), and its replica trajectory shadows the oracle's from slightly above, reflecting the safety margins. It violates only 2 of 58 windows (3.4%)—the flash-crowd onset, which no causal policy can anticipate, and one marginal TPOT window—at 4.32 GPU·h. The reactive baseline spends slightly more (4.42 GPU·h) yet violates 27.6% of windows: every ramp segment begins with violations, and its multiplicative corrections oscillate (Figure 3, bottom). Static peak provisioning is violation-free but costs 10.95 GPU·h, 2.5× PACO; static mean provisioning is cheap but unusable (69% violations). Relative to the clairvoyant oracle (3.41

GPU·h, 1.7%), PACO pays a 27% cost premium for causality and safety margins. A second workload realization (different random seed) reproduces the picture: 5.2% violations for PACO—equal to the oracle on that trace—versus 24.1% for reactive at matched cost.

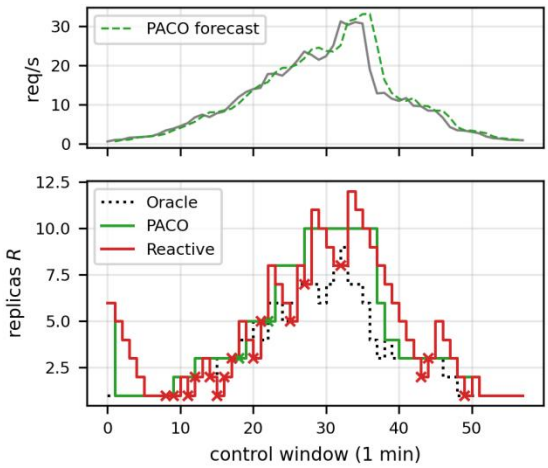

Figure 3 Closed-loop Timeline on the Conversation Workload. Top: Actual Per-window Request Rate and PACO's (upward-biased) Forecast. Bottom: Replica Allocation of Oracle, PACO, and Reactive; Crosses Mark SLO-violating Windows

5.3 Ablation Study

Table 2 Ablations (Violation Rate / Cost in GPU·h)

Variant	conv viol.	conv cost	code viol.	code cost
PACO (full)	3.4%	4.32	23.9%	6.78
− forecaster	6.9%	4.26	37.0%	6.25
− guard	3.4%	4.30	23.9%	6.62
Oracle	1.7%	3.41	2.2%	6.83

Table 2 isolates the contribution of each component. Removing the forecaster doubles violations on the conversation workload (3.4% → 6.9%) and raises them from 23.9% to 37.0% on the spikier code workload: acting on last-window observations re-introduces exactly the one-window lag that penalizes reactive scaling, while costing almost the same. Removing the guard leaves the two primary workloads nearly unchanged—the predictive path alone already handles them—but on the second conversation realization the guard lowers violations from 6.9% to 5.2%, and by construction it bounds any run of consecutive violating windows caused by forecast or model drift; it is cheap insurance whose premium is near zero when predictions are good.

5.4 Long-Context, Spiky Workload

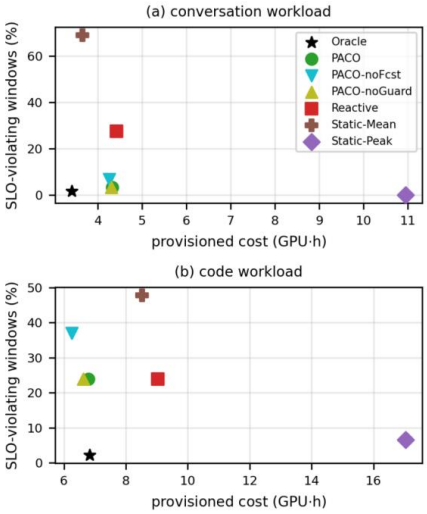

Figure 4 Cost vs. Reliability of All Policies on (a) The Conversation and (b) The Code Workload. Down-left is Better; PACO Sits Closest to the Oracle on the Achievable Frontier

The code workload stresses both design assumptions: prompts are 6× longer at the tail, and the arrival process contains minute-scale spikes in which the rate jumps by up to 26× relative to the previous window (e.g., 0.9 → 16.1 req/s), which no history-based forecaster can anticipate. Figure 4(a-b) shows the outcome. PACO matches the reactive baseline's reliability (23.9% violating windows each) at 24.9% lower cost (6.78 vs. 9.03 GPU·h) and dominates both static policies; the residual gap to the oracle (2.2%) is concentrated almost entirely in the unpredictable spike windows. The experiment also exposes why the token budget must be part of the action space: the oracle selects B = 512 in 45 of 46 windows, because with 7 k-token prompts any larger chunked-prefill budget lets prefill inflate decode iterations past the TPOT SLO—a regime documented for chunked-prefill schedulers [6]. A replica-count-only controller could not express this configuration at any price.

5.5 Overheads and Limitations

PACO's runtime footprint is small: monitoring is O(1) per request, and a full decision over 42 configurations takes ≈0.1 s per minute on one CPU core. The main capital cost is the one-time profiling corpus (3060 window replays per hardware/model pair), which parallelizes trivially. Our study inherits the usual limits of calibrated simulation—the cost model abstracts kernel-level effects, and we assume homogeneous GPUs and a two-knob action space; extending the search to model parallelism, GPU frequency [4], or phase-split pools [3,15], and validating on a physical testbed, are natural next steps. The code-workload results also delineate a fundamental boundary: when arrivals are statistically unpredictable, prediction degrades gracefully to reactive behavior, and further gains must come from spike-absorbing mechanisms (admission control, serverless burst capacity) rather than better forecasting [16].

6 RELATED WORK

6.1 LLM Serving Systems

Orca introduced iteration-level continuous batching [5]; vLLM added paged KV-cache management [9]; Sarathi-Serve characterized the chunked-prefill throughput–latency trade-off that our B knob controls [6]. Splitwise and DistServe disaggregate prefill and decode across machine pools [3,15], and AlpaServe multiplexes models via parallelism [17]. These systems optimize a deployment's data path; PACO is complementary and decides, online, which deployment to run.

6.2 SLO-aware Serving and Autoscaling

Clipper [18], MArk [16], Clockwork [19], and INFaaS pioneered SLO-aware DNN serving [20], including predictive scaling for per-request models whose latency is nearly deterministic. LLM serving breaks their assumptions: latency is a coupled, batch- and length-dependent distribution, which is why PACO predicts windowed tail percentiles with a learned model instead of composing per-request estimates. Closest to us, DynamoLLM reconfigures LLM clusters (instances, parallelism, frequency) for energy under SLOs using expert-derived load thresholds [4]; PACO instead learns the workload-conditioned latency surface, adds explicit rate forecasting, and targets the chunked-prefill budget, and we quantify each component's contribution by ablation.

6.3 Performance Prediction and Configuration Search

Ernest and CherryPick predict job performance to pick cloud configurations offline [7,8]; Vidur shows that calibrated simulation plus learned operator models can explore LLM deployment spaces cheaply [13], a methodology our profiling stage adopts. PACO differs in closing the loop: it couples such predictors with forecasting and feedback to reconfigure a live cluster window by window.

7 CONCLUSION

We presented PACO, a predictive auto-configuration framework for LLM inference serving under tail-latency SLOs that unifies workload forecasting, a learned windowed-percentile performance model, constrained minimum-cost search over replica count and chunked-prefill budget, and a reactive guard. Driven by production Azure traces in an iteration-level cluster simulation, PACO cut SLO-violating windows by 8× versus a reactive autoscaler at no extra cost, saved 60.5% versus peak provisioning, and approached a clairvoyant oracle within 27% cost, while its ablations quantified the value of forecasting and of engine-level knobs—small chunked-prefill budgets prove indispensable for long-context workloads. We release our simulator, profiling corpus, and controller code to support reproduction.

COMPETING INTERESTS

The authors have no relevant financial or non-financial interests to disclose.

REFERENCES

[1] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. Advances in Neural Information Processing Systems (NeurIPS), 2017.
[2] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners. Advances in Neural Information Processing Systems (NeurIPS), 2020, 159: 1877-1901.
[3] Patel P, Choukse E, Zhang C, et al. Splitwise: Efficient generative LLM inference using phase splitting. Proc. 51st ACM/IEEE Int. Symposium on Computer Architecture (ISCA), 2024: 118-132.
[4] Stojkovic J, Zhang C, Goiri Í, et al. DynamoLLM: Designing LLM inference clusters for performance and energy efficiency. Proc. IEEE Int. Symposium on High-Performance Computer Architecture (HPCA), 2025: 1348-1362.
[5] Yu G I, Jeong J S, Kim G W, et al. Orca: A distributed serving system for transformer-based generative models. Proc. 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2022: 521-538.
[6] Agrawal A, Kedia N, Panwar A, et al. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. Proc. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2024: 117-134.
[7] Alipourfard O, Liu H H, Chen J, et al. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. Proc. 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2017.
[8] Venkataraman S, Yang Z, Franklin M, et al. Ernest: Efficient performance prediction for large-scale advanced analytics. Proc. 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2016.
[9] Kwon W, Li Z, Zhuang S, et al. Efficient memory management for large language model serving with PagedAttention. Proc. 29th ACM Symposium on Operating Systems Principles (SOSP), 2023: 611-626.
[10] Holt C C. Forecasting seasonals and trends by exponentially weighted moving averages. International Journal of Forecasting, 2004, 20(1): 5-10.
[11] Friedman J H. Greedy function approximation: A gradient boosting machine. The Annals of Statistics, 2001, 29: 1189-1232.
[12] Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 2011, 12: 2825-2830.
[13] Agrawal A, Kedia N, Mohan J, et al. Vidur: A large-scale simulation framework for LLM inference. Proceedings of Machine Learning and Systems (MLSys), 2024, 6: 351-366.
[14] Touvron H, Lavril T, Izacard G, et al. LLaMA: Open and efficient foundation language models. LLaMA: Open and Efficient Foundation Language Models, 2023.
[15] Zhong Y, Liu S, Chen J, et al. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. Proc. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2024.
[16] Zhang C, Yu M, Wang W, et al. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. Proc. USENIX Annual Technical Conference (ATC), 2019: 1049-1062.
[17] Li Z, Zheng L, Zhong Y, et al. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. Proc. 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2023.
[18] Crankshaw D, Wang X, Zhou G, et al. Clipper: A low-latency online prediction serving system. Proc. 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2017: 613-627.
[19] Gujarati A, Karimi R, Alzayat S, et al. Serving DNNs like clockwork: Performance predictability from the bottom up. Proc. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2020.
[20] Romero F, Li Q, Yadwadkar N J, et al. INFaaS: Automated model-less inference serving. Proc. USENIX Annual Technical Conference (ATC), 2021.