Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

GPU Memory Usage Prediction for Generative AI Serving Pipelines with Queue, Latency, and Utilization Signals

Lee Ji-su
Computer Science, Postech, Pohang, GB, South Korea
jisu.lee_pro@outlook.com

DOI: 10.69987/JACS.2026.60701

Keywords

generative AI serving; GPU memory prediction; diffusion pipelines; queueing signals; latency-aware forecasting; GPU utilization; GenTD26; serverless inference.

Abstract

Generative AI serving pipelines combine prompt processing, model and adapter selection, iterative generation, queueing, and GPU-resident state, so their memory footprint depends on more than request volume alone. This study investigates next-window GPU memory prediction with cross-layer signals from Alibaba GenTD26. The public request trace is transformed into a deterministic pod-window benchmark containing 12,253 active 10-minute samples constructed from 26,823 request records and 143 reproducible pseudo-pods. The feature space combines request complexity, queue pressure, execution and pipeline latency, GPU duty-cycle proxy, pod-memory-utilization proxy, and recent resident-memory lags. Six predictors are evaluated with chronological train, validation, and test partitions. Stochastic-gradient regression provides the lowest test RMSE at 2.588 GiB, with MAE 1.903 GiB and R^2 0.608; Ridge and ExtraTrees follow closely. A feature ablation reduces DecisionTree RMSE from 3.469 GiB with request-only variables to 2.889 GiB with the full cross-layer signal set, a 16.7% reduction. Error increases from the 10-minute to the 30-minute horizon, emphasizing the value of near-term forecasts for admission control, pod selection, and warm-state management. The results show that queue, latency, utilization, and recent memory state jointly provide a practical basis for lightweight GPU-memory forecasting in generative serving pipelines.

1. Introduction

Generative AI has changed online inference from a largely stateless request–response operation into a multi-stage service pipeline. Transformer models introduce long-lived key–value state and dynamic batching, while diffusion models execute iterative denoising and often combine a base model with lightweight adapters and conditioning modules [1]–[5]. These components compete for accelerator memory before, during, and after request execution. As a result, memory availability constrains feasible concurrency, routing, caching, and scale-out decisions even when compute capacity remains available.

Modern serving systems increasingly treat memory management as a first-order systems concern. Orca coordinates iterative generation at the request level, PagedAttention reduces fragmentation in large-language-model serving, and DeepSpeed-Inference combines parallelism and kernel optimization for large models [6]–[8]. ServerlessLLM further shows that model placement and fast checkpoint loading are central to elastic inference, while FlexPipe addresses dynamic pipeline adaptation in fragmented serverless clusters [9], [10]. Across these systems, a scheduler benefits from knowing not only

how busy a GPU is, but also how much memory a pipeline is likely to occupy in the next decision interval.

Application metadata alone is insufficient for that forecast. A request reveals task type, prompt length, image count, inference steps, and adapter count at admission time, but the memory state observed a few minutes later also reflects queue growth, cold loading, resident weights, recent latency, and device utilization. Prediction-serving systems such as Clipper, InferLine, TensorFlow Serving, and Nexus established the importance of latency-aware provisioning and coordinated execution [11]–[14]. GPU-memory forecasting extends this line of work by using the interaction among application, middleware, and infrastructure signals to estimate whether a request can be placed safely.

The same interaction appears at cluster scale. Borg, Kubernetes, and Quasar emphasize resource-aware placement and quality-of-service constraints [15]–[17], while recent GPU-cluster traces document heterogeneous workloads and fragmentation effects that are not visible in CPU-centric traces [18], [19]. Production analysis of diffusion serving has made the cross-layer relationship especially clear: model popularity, component combinations, loading overhead, utilization, and GPU memory vary together across the service stack [20]. Alibaba GenTD26 exposes this top-down structure through request, queue, latency, pod-memory, GPU-duty, and GPU-memory tables that can be correlated at the container level [21].

This study asks whether a compact forecasting pipeline can recover useful next-window memory structure from request complexity and recent service-state signals. The experiment uses the GenTD26 request trace to construct a deterministic memory proxy at a 10-minute pod-window resolution, then compares six lightweight regressors under a chronological split. The proxy formulation is explicit: it represents model size, task type, adapter count, generation complexity, latency, queue pressure, utilization, and decaying resident state. It is therefore suitable for studying signal value and temporal generalization without treating the proxy as direct allocator telemetry.

The paper makes three contributions. First, it formulates GPU-memory prediction as a cross-layer forecasting task in which application descriptors are augmented by queue, latency, utilization, and resident-state history. Second, it quantifies the value of those signal groups through model comparison, feature ablation, forecast-horizon analysis, memory-band errors, and feature importance. Third, it presents an operational interpretation of the forecasts for short-horizon admission control, warm-state retention, and memory headroom management.

2. Literature Review

2.1 Generative model serving and memory management

The serving behavior of generative models follows directly from their computational structure. Attention-based transformers [1] create sequence-dependent state, whereas diffusion models [2]–[4] repeat denoising operations over a configurable number of steps. LoRA adds small trainable matrices to a shared base model [5], which is efficient for adaptation but creates a serving problem when many adapters must be loaded, merged, cached, or evicted. These properties make a request's memory footprint sensitive to both static descriptors and the recent history of model residency.

Serving systems address different parts of this problem. Orca interleaves iterations across requests to improve utilization [6], PagedAttention organizes key–value memory in non-contiguous blocks [7], and DeepSpeed-Inference combines tensor, pipeline, and kernel optimizations [8]. ServerlessLLM reduces cold-start cost through locality-aware checkpoint storage and loading [9], while FlexPipe adapts pipeline structure when serverless clusters are fragmented [10]. These systems motivate a forecast that can be evaluated quickly and incorporated into scheduling before a device reaches a hard memory limit.

Diffusion serving adds component heterogeneity to the same resource problem. A production study by Lin et al. links request skew, base models, LoRA adapters, loading latency, utilization, and memory consumption across application, middleware, and

hardware layers [20]. That evidence supports the central premise of this paper: future GPU memory is not merely an autoregressive telemetry signal; it is the result of a service pipeline whose workload composition and queue state are observable.

2.2 Cross-layer serving, scheduling, and cluster traces

Prediction-serving research has long connected latency objectives with resource allocation. Clipper focuses on low-latency model serving, InferLine provisions multi-stage pipelines against latency targets, TensorFlow Serving standardizes model lifecycle management, and Nexus coordinates GPU execution for deep neural network workloads [11]–[14]. Their designs differ, but each shows that performance emerges from interactions among request arrival patterns, batching, model placement, and hardware state.

Cluster managers provide the complementary placement perspective. Borg and Kubernetes organize heterogeneous services around resource requests and scheduling constraints [15], [16], while Quasar uses classification and resource allocation to satisfy quality-of-service goals [17]. GPU-specific studies reveal additional challenges: production machine-learning clusters combine heterogeneous devices and workload classes [18], and GPU sharing can lose capacity through fragmentation even when aggregate free resources appear adequate [19]. These findings make memory headroom a placement variable rather than a passive monitoring metric.

GenTD26 is particularly relevant because its data organization follows the same top-down path. The dataset describes application-level requests, middleware queue and latency metrics, and infrastructure-level pod memory, GPU duty cycle, and GPU memory; shared container identifiers support cross-table correlation [21]. The present experiment adopts that conceptual structure and uses the request trace to create a compact pod-window benchmark in which the same signal families can be evaluated together.

2.3 GPU demand forecasting and risk-aware capacity decisions

Time-series forecasting offers a broad set of statistical and neural methods for workload planning. DeepAR learns autoregressive probabilistic forecasts [26], and the Temporal Fusion Transformer combines multi-horizon forecasting with interpretable temporal attention [27]. General forecasting practice also emphasizes temporal validation, meaningful baselines, and horizon-dependent error analysis [28]. These principles are directly relevant to serving traces, where random data splits can leak local temporal state and overstate deployment performance.

Recent GPU-capacity studies place forecasting within operational decisions. Multi-horizon demand forecasting on Alibaba GPU traces finds that autoregressive structure dominates short horizons while semantic context becomes more useful farther ahead [22]. Risk-bounded oversubscription uses conformal demand envelopes to connect forecasts to capacity limits [23], and profit-aware spot-GPU admission control combines chronological forecasting with cost-sensitive acceptance policies [24]. Cross-cloud capacity forecasting further highlights workload and topology distribution shift as a practical generalization problem [25]. Together, these studies suggest that point accuracy is only one layer of the problem: the forecast must also support a safe action under changing demand.

The present study occupies a narrower but complementary position. Rather than forecasting cluster-wide GPU counts, it predicts next-window memory for an active service group. The model set is deliberately lightweight and implemented with standard machine-learning components [29], including extremely randomized trees [30]. This choice keeps the contribution centered on the signal hierarchy. Explicit transformations, fixed splits, and compact feature sets also reduce the hidden coupling that often accumulates in production machine-learning pipelines [31].

3. Method

3.1 Dataset and prediction task

The experiment uses the public Alibaba GenTD26 request trace, lora_request_trace.csv. GenTD26 was designed for top-down analysis of generative serving and includes request characteristics together with queue, pipeline-latency, pod-memory, GPU-duty, and GPU-memory data products [20], [21]. The request table used here contains timestamp, prediction type, request status, execution time, group identifier, prompt and negative-prompt lengths, images per prompt, inference steps, checkpoint-model identifier, and number of LoRA adapters.

The prediction unit is an active pseudo-pod in a 10-minute window. Because the request table does not expose the infrastructure container identifier on each request row, a stable hash of group and checkpoint identifiers assigns every request to one of 143 pseudo-pods. This mapping creates reproducible pod-local histories while remaining distinct from the production scheduler's actual placement. The target is next-window derived GPU memory, expressed in GiB, and should be interpreted as a deterministic benchmark proxy for resident memory rather than direct device telemetry. Table 1 summarizes the task definition and experimental scope.

Table 1. Dataset and prediction-task definition.

Item	Value
Dataset family	Alibaba GenTD26 top-down generative AI serving trace
Input table	lora_request_trace.csv
Request records	26,823
Observation period	2024-11-15 16:57:50 to 2024-12-08 17:33:57
Evaluation unit	Active pseudo-pod × 10-minute window
Benchmark samples	12,253 pod-window observations
Target	Next-window derived GPU memory in GiB
Forecast horizons	10, 20, and 30 minutes
Evaluation design	Chronological train, validation, and test partitions

3.2 Request preprocessing and window aggregation

Timestamps are parsed in chronological order, numeric missing values are filled with training-compatible median values, and missing negative-prompt lengths are set to zero. Request status is represented through success, failure, pending, and processing indicators. The data are then grouped by pseudo-pod and the 10-minute floor of the request timestamp. Aggregates include request count, success rate, prompt statistics, images per request, inference steps, LoRA count, task-type proportions, execution-time summaries, and cyclical hour-of-day variables.

The raw request distribution is dominated by text-to-image generation, with smaller image-to-image and inpainting segments. Execution time and generation complexity are right-skewed, which makes median and high-percentile summaries more informative than the mean alone. Table 2 reports the principal characteristics measured after loading and cleaning the request table.

Table 2. Characteristics of the GenTD26 request trace used in the experiment.

Characteristic	Measured value
Rows	26,823
Time range	2024-11-15 16:57:50 to 2024-12-08 17:33:57
Unique groups	4,247
Unique base-model IDs	86
Request types	TXT_2_IMG=24,438; IMG_2_IMG=2,199; INPAINTING=186
Request status	SUCCEED=26,392; FAILED=398; PENDING=30; PROCESSING=3
Execution time	median=23.0 s; p95=69.0 s
Inference steps	median=30; p95=40
Images per request	median=1; p95=8
LoRA adapters	mean=0.181; p95=1; max=6

3.3 Derived memory benchmark and cross-layer signals

For each request, the derived memory proxy combines a model-specific base component with increments for task type, LoRA count, prompt complexity, image count, inference steps, execution latency, and request status. The sum is bounded to a practical accelerator-memory interval. At the pod-window level, the maximum active-request proxy is blended with a decaying resident-state term, reflecting the fact that base weights and adapters can remain in memory after a request completes. In compact form, the construction is

$$M_{request} = \mathrm{clip}(M_{model} + M_{task} + M_{adapter} + M_{prompt} + M_{generation} + M_{latency}, 12, 31.5),$$

$$M_{pod,t} = \mathrm{resident_blend}(\max M_{request} \text{ in window } t, M_{pod,t-1}).$$

Queue size is derived from concurrent demand within a window, queue response time combines demand and request complexity, and pipeline latency combines queue response with execution time. GPU duty-cycle and pod-memory-utilization proxies are functions of demand, steps, image count, LoRA count, and resident memory. Pod-local lags are then computed for memory and utilization. Table 3 lists the construction of each signal family, and Figure 1 shows how application, middleware, and infrastructure-derived observations feed the next-window target.

Table 3. Deterministic construction of the pod-window benchmark.

Component	Columns	Construction
Pod identity	pseudo_pod	Stable hash of group and checkpoint identifiers; P000–P142
Windowing	window	10-minute floor of request timestamp
Queue	queue_size_proxy; queue_rt_proxy_ms	Window demand and request-complexity pressure

Component	Columns	Construction
Latency	pipeline_latency_proxy_ms	Queue-response component plus execution-time component
Utilization	gpu_duty_cycle_proxy; pod_memory_util_proxy	Demand, generation complexity, and resident-memory state
Resident memory	resident_signal	Pod-local exponentially decayed active-request memory
Target	target_h1_memory_gib	Derived memory in the next active pod-window
Lags	memory_lag_1/2/3; util_lag_1/2	Past pod-local memory and duty-cycle values

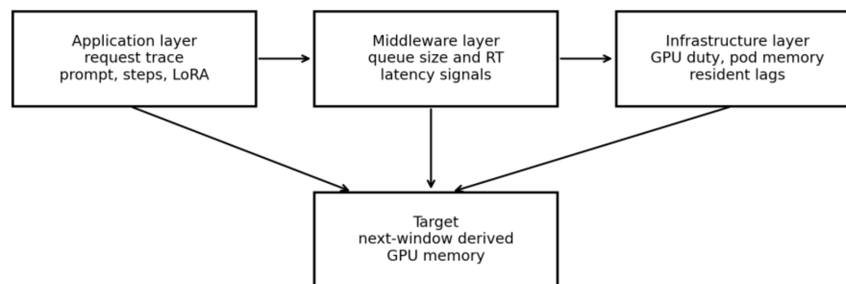

Figure 1. Top-down signal construction for next-window GPU-memory prediction.

3.4 Feature sets, temporal split, and forecast horizons

Four nested feature sets isolate the contribution of each layer. The request-only set contains request counts, success rate, prompt statistics, image count, inference steps, LoRA count, task-type rates, hour-of-day variables, and pod identity. The request-plus-queue set adds queue size and queue-response proxies. The request–queue–latency set additionally includes mean and p95 execution time, pipeline latency, and rolling latency. The full set adds GPU duty cycle, pod-memory utilization, three memory lags, and two utilization lags.

The principal target, target_h1_memory_gib, is the next active 10-minute pod-window. Two additional targets represent 20-minute and 30-minute horizons. Samples are sorted by time and partitioned chronologically: the first 70% for training, the next 15% for validation, and the final 15% for testing. Table 4 gives the realized date ranges and sample counts. This design is intentionally stricter than random splitting because pod-local lags and recurring traffic patterns would otherwise leak nearby future conditions into training.

Table 4. Chronological split used for evaluation.

Split	Rows	Start window	End window
Train	7,818	2024-11-15 18:40:00	2024-12-02 18:10:00
Validation	2,940	2024-12-02 18:20:00	2024-12-04 23:20:00
Test	1,495	2024-12-04 23:30:00	2024-12-08 17:00:00

3.5 Models and evaluation metrics

Six predictors span naive, linear, and tree-based approaches. MeanBaseline uses the training-set mean, and Lag1Baseline copies the most recent memory lag. Ridge and stochastic-gradient descent (SGD) are fitted after median imputation and standardization. DecisionTree provides a depth-limited nonlinear model, while ExtraTrees averages randomized trees and captures interactions without extensive tuning [29], [30]. The fixed configurations are listed in Table 5.

Table 5. Model configurations.

Model	Configuration
MeanBaseline	Training-set mean target
Lag1Baseline	Copies memory_lag_1_gib
Ridge	alpha=2.0; median imputation; standard scaling
SGD	squared_error; alpha=0.0005; max_iter=1500; standard scaling
DecisionTree	max_depth=8; min_samples_leaf=12
ExtraTrees	n_estimators=12; max_depth=10; min_samples_leaf=8

Performance is measured with mean absolute error (MAE), root mean squared error (RMSE), coefficient of determination (R^2), and mean absolute percentage error (MAPE) with a nonzero denominator guard. MAE expresses average reservation error in GiB, RMSE emphasizes large misses, R^2 measures improvement over a constant predictor, and MAPE provides a relative scale. The combination is useful because high-memory windows have greater operational cost even when they are uncommon.

4. Results and Discussion

4.1 Benchmark profile and temporal behavior

The transformation produces 12,253 active pod-window samples spanning 1,476 distinct 10-minute windows and 143 pseudo-pods. Derived memory has a mean of 18.996 GiB, a 95th percentile of 27.119 GiB, and a maximum of 31.5 GiB. Figure 2 plots the aggregate mean and p95 across time. The visible bursts and changes in high-percentile memory confirm that a single static reservation cannot represent the test period well.

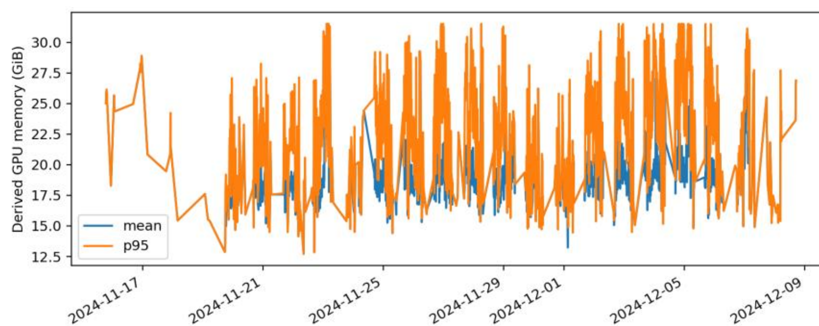

Figure 2. Aggregate derived GPU memory over active 10-minute pod-windows.

The series also illustrates why temporal validation matters. High-memory intervals cluster around changes in workload composition and service pressure, while the decaying resident-state term carries part of each burst into later windows. A random split would place neighboring states in both training and testing; the chronological split instead requires each model to extrapolate from earlier operation into a later demand regime.

4.2 Main model comparison

Table 6 presents the test results using the full signal set. SGD achieves the lowest RMSE at 2.588 GiB and R^2 of 0.608. ExtraTrees has the lowest MAE and MAPE by a small margin, while Ridge remains within 0.020 GiB of the best RMSE. Figure 3 makes the ordering by RMSE explicit.

Table 6. Model comparison on the chronological test split.

Model	MAE (GiB)	RMSE (GiB)	R^2	MAPE (%)	Fit+predict (s)
SGD	1.903	2.588	0.608	9.450	0.041
Ridge	1.918	2.608	0.602	9.475	0.025
ExtraTrees	1.895	2.610	0.602	9.401	0.175
DecisionTree	2.050	2.889	0.512	10.146	0.104
Lag1Baseline	2.437	3.396	0.326	12.422	0.001
MeanBaseline	3.046	4.287	−0.074	14.126	0.002

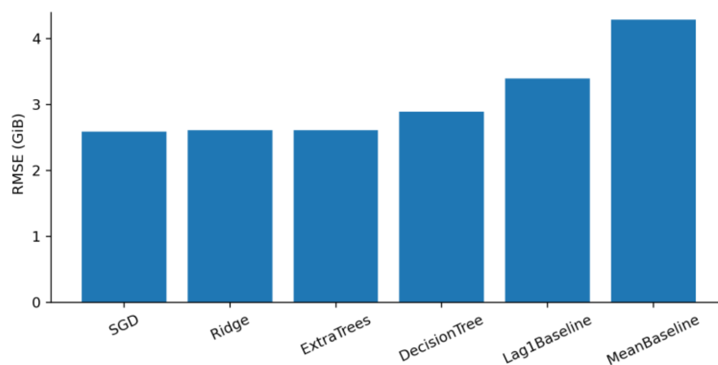

Figure 3. RMSE comparison across forecasting models

The gap between the trained models and Lag1Baseline is operationally meaningful. Copying the most recent memory state captures persistence, but it misses changes in request mix, queue pressure, and generation complexity. The mean baseline performs worst and yields a negative R^2, showing that the later test period is not well represented by the training-period average. A scheduler that reserves memory from a fixed historical mean would therefore alternate between excess reservation in light periods and insufficient headroom in heavy periods.

The closeness of SGD, Ridge, and ExtraTrees indicates strong additive structure in the benchmark. Recent memory, pod-memory utilization, rolling latency, and request complexity explain much of the target without a large neural model. The result is consistent with the broader observation that short-horizon GPU demand is often dominated by autoregressive state [22], while richer context becomes more valuable as the decision horizon expands.

4.3 Contribution of queue, latency, utilization, and lag signals

The nested ablation in Table 7 tests the central hypothesis directly. With DecisionTree held fixed, request-only variables produce RMSE 3.469 GiB. Queue variables lower RMSE to 3.123 GiB, latency variables reduce it to 2.907 GiB, and the full signal set reaches 2.889 GiB. The request-only to full-signal reduction is 0.580 GiB, or 16.7%. Figure 4 visualizes the stepwise improvement.

Table 7. Feature ablation with the DecisionTree model.

Feature set	Features	MAE (GiB)	RMSE (GiB)	R^2	MAPE (%)
Full signals	25	2.050	2.889	0.512	10.146
Request + queue + latency	18	2.109	2.907	0.506	10.432
Request + queue	14	2.244	3.123	0.430	10.879
Request only	12	2.466	3.469	0.296	11.620

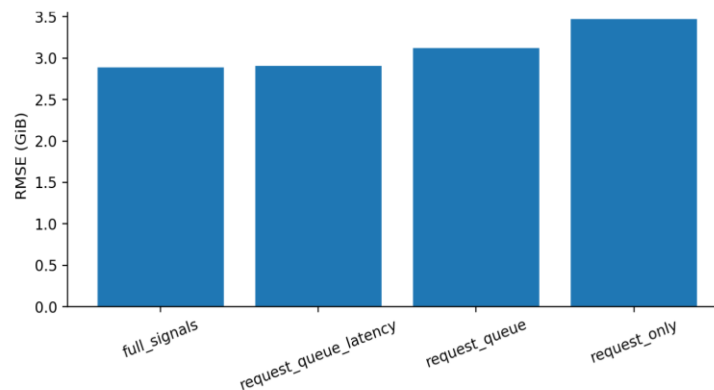

Figure 4. Feature ablation showing the effect of cross-layer signals on RMSE.

Queue length does not allocate GPU memory directly, but it signals demand concentration and likely batching or warm-route reuse. Latency similarly summarizes several hidden service states, including heavier generation, cold loading, and contention. Utilization and memory lags then describe the infrastructure state carried from recent windows.

The ablation therefore supports a causal interpretation of the feature hierarchy: request fields describe incoming work, queue and latency describe the service path, and utilization and lags describe the resource state left by recent work.

This hierarchy also provides a deployment trade-off. Admission-time decisions can use request-only or request-plus-queue variables before complete telemetry is available. Decisions made a few seconds later can incorporate recent pod utilization and memory lags for better accuracy. The full model is most appropriate for short-horizon routing and reservation, whereas earlier feature sets remain useful when low decision latency is more important than minimum forecast error.

4.4 Forecast horizon and conditional error

Table 8 shows monotonic degradation as the horizon increases. DecisionTree RMSE rises from 2.889 GiB at 10 minutes to 3.260 GiB at 20 minutes and 3.506 GiB at 30 minutes. R^2 falls from 0.512 to 0.285. The pattern is expected because longer horizons contain more unobserved arrivals and larger changes in resident state.

Table 8. Forecast-horizon comparison using DecisionTree and the full signal set.

Horizon (min)	MAE (GiB)	RMSE (GiB)	R^2	MAPE (%)
10	2.050	2.889	0.512	10.146
20	2.386	3.260	0.376	11.741
30	2.553	3.506	0.285	12.530

The horizon result positions the predictor as a near-term control signal. A 10-minute forecast can guide pod selection, warm-state retention, and immediate memory admission. A 30-minute forecast remains useful for coarse planning but should be paired with a wider reserve margin or a probabilistic envelope, consistent with risk-aware approaches to GPU oversubscription [23] and supply–demand admission control [24].

Error also varies by target-memory level. Table 9 shows that most test observations fall in the 16–24 GiB band, where SGD achieves RMSE 2.117 GiB. The 24–40 GiB band has RMSE 4.235 GiB, reflecting rarer heavy windows and behavior near the upper bound. This concentration argues for asymmetric operating margins: a tighter reserve is reasonable for mid-range predictions, while high-memory forecasts should trigger a larger safety buffer.

Table 9. Test error by derived target-memory band for SGD.

Target memory band	N	MAE (GiB)	RMSE (GiB)	MAPE (%)
(12.0, 16.0]	189	1.962	2.280	13.057
(16.0, 24.0]	1,070	1.574	2.117	8.257
(24.0, 40.0]	236	3.345	4.235	11.968

Figure 5 compares predicted and actual memory for the best-RMSE model. Predictions follow the central trend but compress some extreme values. Figure 6 shows a residual distribution centered near zero with a longer negative tail, indicating occasional underprediction of high-memory windows. The point forecast is therefore best used as a ranking and reservation input rather than an exact allocator.

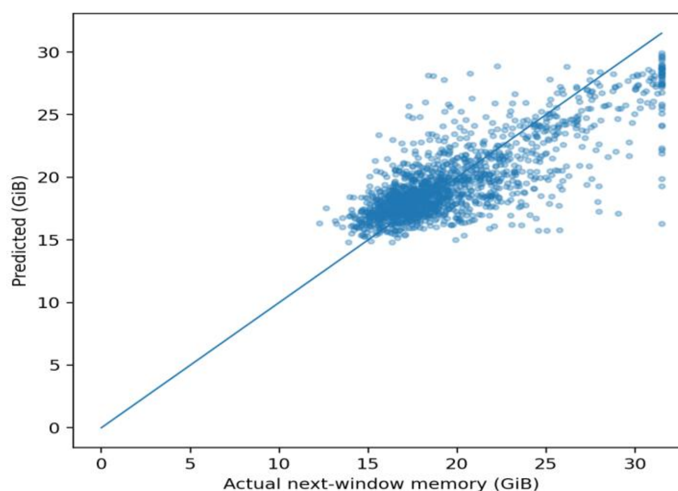

Figure 5. Predicted versus actual next-window derived memory for SGD.

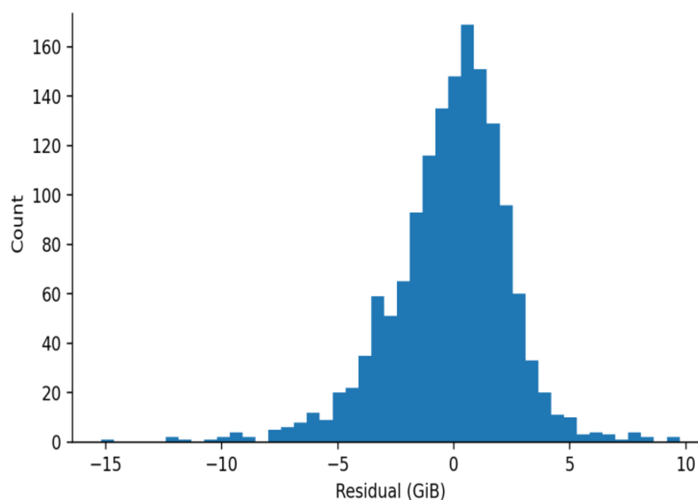

Figure 6. Residual distribution for the SGD model.

4.5 Feature importance and operational implications

Table 10 lists the ten largest ExtraTrees importances. The previous-window memory lag ranks first, followed by pod-memory utilization and rolling latency. Additional memory lags, LoRA count, image count, pipeline latency, and GPU duty cycle also appear in the leading group. Figure 7 displays the broader ranking.

Table 10. Top ten ExtraTrees feature importances.

Feature	Importance
memory_lag_1_gib	0.2811
pod_memory_util_proxy	0.1517

Feature	Importance
rolling_latency_mean_3	0.1198
mean_negative_prompt_length	0.0929
memory_lag_3_gib	0.0811
memory_lag_2_gib	0.0740
mean_num_lora	0.0325
mean_num_images	0.0277
pipeline_latency_proxy_ms	0.0221
gpu_duty_cycle_proxy	0.0206

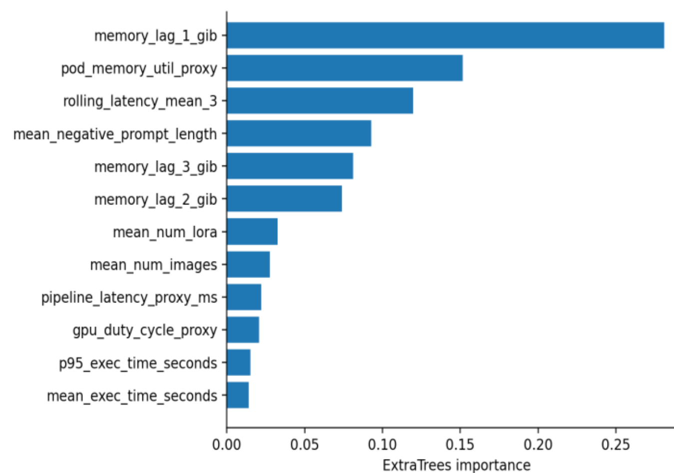

Figure 7. Feature importance for the full-signal ExtraTrees model.

The ranking reinforces the ablation result. Memory is persistent, but persistence alone is incomplete; service-state features explain how the next window departs from the last one. In operational use, the point estimate can be converted into a headroom score, predicted memory divided by device capacity, and combined with a band-dependent safety margin. Requests can then be routed toward pods with sufficient predicted headroom, while high-risk windows can be rejected, delayed, or assigned to a larger device.

The lightweight linear models are attractive for this role because they are fast to update and easy to monitor. More expressive temporal models such as DeepAR or TFT [26], [27] may become valuable when longer measured histories and multiple correlated pods are available, but the present results suggest that careful signal construction can deliver substantial gains before model complexity is increased.

5. Limitations

The first limitation concerns target semantics. The benchmark predicts a deterministic derived memory proxy constructed from the request trace; it is not a measurement of allocator-level GPU memory. The experiment therefore supports conclusions about signal structure, feature value, and temporal forecasting behavior, but the absolute error values should not be interpreted as production telemetry accuracy. A direct extension should join request, queue, latency, pod, and GPU tables through the

container identifier provided by GenTD26 [21] and repeat the analysis with observed GPU-memory values.

Second, pseudo-pods are stable analytical groups rather than reconstructed production placements. They create local temporal histories and make lag features possible, but they do not reproduce the scheduler's routing, cache affinity, or eviction policy. This limitation is especially relevant when the same model can move among devices or when multiple models share one GPU.

Third, the sample includes active pod-windows. Long idle periods and memory retained without a request are not fully observed from the request table alone. Infrastructure telemetry would allow a complete panel of active and idle windows and would make it possible to evaluate unnecessary reservations during extended idle intervals.

Fourth, the models provide point predictions rather than calibrated intervals. High-memory windows have larger errors, so a production controller should estimate uncertainty with quantile regression, conformal prediction, or a probabilistic forecasting model. Risk envelopes are particularly important when the cost of underprediction is an out-of-memory failure rather than a symmetric numerical error [23].

Finally, the study evaluates forecasts rather than end-to-end control outcomes. A scheduler simulation or online experiment is needed to measure memory waste, request rejection, queueing delay, cold starts, and avoided out-of-memory events. Cross-cloud or cross-topology testing would also determine how well the feature relationships transfer under distribution shift [25].

6. Conclusion

This paper presented a cross-layer approach to next-window GPU-memory prediction for generative AI serving pipelines. Using 26,823 GenTD26 request records, the experiment formed 12,253 active 10-minute pod-window samples and combined request complexity with queue, latency, utilization, and resident-memory history. SGD achieved MAE 1.903 GiB and RMSE 2.588 GiB on the chronological test period, while the feature ablation showed a 16.7% RMSE reduction from request-only variables to the full signal set.

The findings support a practical systems conclusion: memory forecasting should not be treated as an isolated univariate telemetry task. Application features explain what is arriving, middleware features describe service pressure, and infrastructure-derived history describes what remains resident. Their combination improves short-horizon prediction and gives schedulers a usable signal for pod selection, warm-state management, and memory admission.

The next step is to replace the proxy target with measured pod GPU memory and evaluate calibrated safety margins inside a scheduling loop. That extension would connect forecast accuracy to the operational outcomes that matter most: higher accelerator utilization, fewer out-of-memory failures, lower cold-start cost, and stable latency under bursty demand.

References

[1] A. Vaswani et al., "Attention Is All You Need," in Advances in Neural Information Processing Systems, 2017, pp. 5998–6008.
[2] J. Ho, A. Jain, and P. Abbeel, "Denoising Diffusion Probabilistic Models," in Advances in Neural Information Processing Systems, 2020, pp. 6840–6851.
[3] J. Song, C. Meng, and S. Ermon, "Denoising Diffusion Implicit Models," in Proc. International Conference on Learning Representations, 2021.
[4] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," in Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 10684–10695.
[5] E. J. Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," in Proc. International Conference on Learning Representations, 2022.
[6] G.-I. Yu et al., "Orca: A Distributed Serving System for Transformer-Based Generative Models," in Proc. 16th USENIX Symposium on Operating Systems Design and Implementation, 2022, pp. 521–538.

[7] W. Kwon et al., “Efficient Memory Management for Large Language Model Serving with PagedAttention,” in Proc. 29th ACM Symposium on Operating Systems Principles, 2023, pp. 611–626.
[8] Z. Wen, R. Zhang, and C. Wang, "Optimization of bi-directional gated loop cell based on multi-head attention mechanism for SSD health state classification model," in 2025 6th International Conference on Electronic Communication and Artificial Intelligence (ICECAI), Chengdu, China, 2025, doi: 10.1109/ICECAI66283.2025.11171441.
[9] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, “ServerlessLLM: Low-Latency Serverless Inference for Large Language Models,” in Proc. 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, pp. 135–153.
[10] Y. Lin, S. Peng, C. Lu, C. Xu, and K. Ye, “FlexPipe: Adapting Dynamic LLM Serving Through Inflight Pipeline Refactoring in Fragmented Serverless Clusters,” in Proc. 21st European Conference on Computer Systems, 2026, doi: 10.1145/3767295.3769316.
[11] D. Crankshaw et al., “Clipper: A Low-Latency Online Prediction Serving System,” in Proc. 14th USENIX Symposium on Networked Systems Design and Implementation, 2017, pp. 613–627.
[12] D. Crankshaw et al., “InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines,” in Proc. ACM Symposium on Cloud Computing, 2020, pp. 477–491.
[13] C. Olston et al., “TensorFlow-Serving: Flexible, High-Performance ML Serving,” arXiv:1712.06139, 2017.
[14] H. Shen et al., “Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis,” in Proc. 27th ACM Symposium on Operating Systems Principles, 2019, pp. 322–337.
[15] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-Scale Cluster Management at Google with Borg,” in Proc. European Conference on Computer Systems, 2015, pp. 1–17.
[16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” Communications of the ACM, vol. 59, no. 5, pp. 50–57, 2016.
[17] C. Delimitrou and C. Kozyrakis, “Quasar: Resource-Efficient and QoS-Aware Cluster Management,” in Proc. International Conference on Architectural Support for Programming Languages and Operating Systems, 2014, pp. 127–144.
[18] Q. Weng et al., “MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters,” in Proc. 19th USENIX Symposium on Networked Systems Design and Implementation, 2022, pp. 945–960.
[19] Q. Weng et al., “Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent,” in Proc. USENIX Annual Technical Conference, 2023, pp. 995–1008.
[20] Y. Lin, S. Wu, S. Luo, H. Xu, H. Shen, C. Ma, M. Shen, L. Chen, C. Xu, L. Qu, and K. Ye, “Understanding Diffusion Model Serving in Production: A Top-Down Analysis of Workload, Scheduling, and Resource Efficiency,” in Proc. ACM Symposium on Cloud Computing, 2025, 15 pp., doi: 10.1145/3772052.3772206.
[21] Alibaba Cluster Data, “GenTD26: Generative AI Serving Top-Down Dataset,” cluster-trace-v2026-GenAI, 2026.
[22] S. Zhao, J. Bai, and D. Roberson, “Multi-Horizon GPU Demand Forecasting with Workload Semantics and Operational Risk Curves: An Empirical Study on Alibaba Clusterdata GPU Trace,” Journal of Technology Informatics and Engineering, vol. 4, no. 3, pp. 544–571, Dec. 2025, doi: 10.51903/jtie.v4i3.498.
[23] S. Chen, S. He, and E. Sun, “Risk-Bounded GPU Resource Oversubscription via Conformal Demand Envelopes in Production AI Clusters,” Journal of Advanced Computing Systems, vol. 4, no. 5, pp. 119–134, May 2024, doi: 10.69987/JACS.2024.40509.
[24] S. Zhao, Y. Ren, and X. Chang, “Profit-Aware Spot GPU Admission Control with Cost-Sensitive Loss and Evidence-Grounded Policy Memos for AI Workload Supply-Demand Matching,” Journal of Technology Informatics and Engineering, vol. 5, no. 2, pp. 45–59, Jun. 2026, doi: 10.51903/jtie.v5i2.545.
[25] S. He, X. Chang, and E. Sun, “Cross-Cloud Transfer Learning for AI Training Capacity Forecasting under Workload and Topology Distribution Shift,” Journal of Advanced

Computing Systems, vol. 4, no. 1, pp. 100–120, Jan. 2024, doi: 10.69987/JACS.2024.40108.
[26] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski, “DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks,” International Journal of Forecasting, vol. 36, no. 3, pp. 1181–1191, 2020.
[27] B. Lim, S. Ö. Arık, N. Loeff, and T. Pfister, “Temporal Fusion Transformers for Interpretable Multi-Horizon Time Series Forecasting,” International Journal of Forecasting, vol. 37, no. 4, pp. 1748–1764, 2021.
[28] R. J. Hyndman and G. Athanasopoulos, Forecasting: Principles and Practice, 3rd ed. Melbourne, Australia: OTexts, 2021.
[29] F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.
[30] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely Randomized Trees,” Machine Learning, vol. 63, no. 1, pp. 3–42, 2006.
[31] D. Sculley et al., “Hidden Technical Debt in Machine Learning Systems,” in Advances in Neural Information Processing Systems, 2015, pp. 2503–2511.