LLM-Pilot: SLO-Aware and Cost-Efficient LLM Serving on Public Cloud VM Clusters via Offloading

Jinwoo Kim*, Kihyun Kim*, Hyunsun Chung*, Jihoon Yang*, James J. Kim†, Dong Li‡, Youngjae Kim*,§
*Sogang University, †Soteria Inc., ‡University of California, Merced
{jinwookim, kion777, hchung1652, yangjh, youkim}@sogang.ac.kr, jkim@soteria-sys.com, dli35@ucmerced.edu

***Abstract*—The prohibitive memory footprint of Key-Value (KV) caches in large language models (LLMs) limits batch concurrency and maximum context length under fixed GPU memory, thereby degrading serving throughput and long-context capability. Techniques such as KV cache offloading (KVO) and attention offloading (AO) can relieve GPU memory pressure, but optimizing them in the cloud is difficult because VM instances bundle compute, memory capacity, and (PCIe/network/storage) bandwidth in pre-packaged VM instances. Since offloading shifts the bottleneck across these resources and the optimal trade-off depends on Service Level Objectives (SLOs), selecting a cost-optimal combination of offloading strategies and instance types becomes a high-dimensional decision problem. We propose LLM-Pilot, a cost-aware optimization framework that co-optimizes serving strategies and heterogeneous cloud infrastructure. Using a hierarchical analytical model calibrated with linear regression, LLM-Pilot jointly searches over offloading strategies and instance configurations to maximize cost efficiency while satisfying latency and throughput constraints. Extensive evaluation on AWS shows that LLM-Pilot improves cost efficiency by up to 2.31× over prior methods and finds feasible configurations in resource-intensive regimes where existing baselines fail.**

***Index Terms*—LLM Inference, Cloud Computing, KV Cache Offloading, Attention Offloading, Cost Optimization, SLO**

I. Introduction

Large language models (LLMs) are utilized in various application fields, such as conversational services, document summarization, and retrieval augmented generation (RAG) [1]. The workloads for these applications are largely divided into real-time services and batch services, each requiring distinct service level objectives (SLOs) [2]. While real-time services require low latency and response stability, batch services prioritize throughput and cost-efficiency under relaxed latency constraints. Major LLM service providers, such as OpenAI [3] and Anthropic [4], also reflect these differences by applying different token unit prices for online and batch processing (Table I). Here, the cost structure also varies depending on the workload's SLO even for the same model. Therefore, selecting a resource configuration tailored to the characteristics of each workload is crucial.

Existing research shows in segments where the token generation rate exceeds human reading speed, further reductions in latency do not translate into improved perceived quality for the user. Consequently, using high-end GPUs may offer only limited benefits compared to the increase in cost [5, 6]. Therefore, operating LLM inference in a cloud environment where resources can be selected according to the workload and SLO can be a more rational option over a fixed hardware environment [7]. Since the cloud provides various GPU instances with different performance levels and price points on a pay-as-you-go basis, cost-efficiency can be improved by selecting an instance that matches the workload requirements.

However, even in cloud environments, GPU memory remains a critical constraint. Recently, requirements for sequence lengths have been increasing due to the proliferation of RAG-based applications and requirements for long-context support [8]. At the same time, LLMs must maintain the Key-Value (KV) cache of previous tokens while generating new ones, and as the context lengthens, the memory requirements for the KV cache increase [9]. Consequently, even in cloud environments where various instances can be selected, GPU memory capacity serves as a major factor governing the choice of serving configuration [10].

To address these GPU memory constraints, current LLM serving systems generally employ the following strategies.

- **First, (Scale-up)** utilizes high-end GPUs with large memory capacity. While this approach simplifies configuration, cost-efficiency may degrade due to high unit costs.
- **Second, (Multi-pass)** splits requests into micro-batches for sequential processing to bypass memory limits without extra resources. However, repeatedly executing small batches delays the completion of individual requests and can particularly increase the end-to-end completion time in online workloads.
- **Third, (Offloading)** leverages external resources to supplement VRAM on low-cost instances. Techniques include *KV cache offloading (KVO)*, which moves cache to host memory, and *Attention offloading (AO)*, which delegates attention computation to auxiliary GPUs (local or remote) (§II-B).

Since offloading entails data movement, its effectiveness heavily depends on system characteristics such as bandwidth and latency of PCIe, storage, and networks.

Motivation. Ultimately, determining cost-efficient LLM serving is not reducible to a single strategy but constitutes a complex configuration selection problem. This requires navigating the following coupled factors:

§Corresponding author.

TABLE I
TOKEN PRICING (USD PER 1M TOKENS) FOR MAJOR LLM SERVICE PROVIDERS AS OF NOVEMBER 2025.

Provider	Model	Realtime ($/1M)		Batch ($/1M)	
		Input	Output	Input	Output
OpenAI	GPT-5.1	1.25	10.0	0.625	5.0
Anthropic	Sonnet 4.5	3.0	15.0	1.5	7.5
Google	Gemini-3 Pro	2.0	12.0	1.0	6.0

* Batch services have a 24-hour SLO, whereas Realtime services do not.

- **Strategy & Cost Trade-offs:** The optimal choice between high-end scaling and low-cost offloading shifts dynamically based on workload-specific bottlenecks and cost structures.
- **Bundled Resources:** Cloud instances' rigid coupling of compute, memory, and I/O prevents the independent scaling of specific bottleneck resources.
- **SLO Constraints:** The objective is not merely to minimize cost, but to identify the configuration that *maximizes cost-efficiency* within the feasible region defined by strict latency and throughput SLOs.

Limitations of Prior Work. While prior works have improved cloud-based LLM serving [11, 12, 13], they predominantly focus on isolated strategies, neglecting the comprehensive configuration selection problem in complex cloud infrastructures. For instance, InferSave [12] is limited to single-node host-memory KVO, overlooking multi-VM options like attention offloading. Similarly, Mélange [14] lacks an integrated framework that optimizes offloading strategies under I/O and SLO constraints for cost-efficiency. A unified approach to derive the optimal configuration across these strategies while satisfying strict SLOs remains absent.

Technical Contributions. To address these challenges, we propose LLM-PILOT, a framework for optimizing cloud LLM inference configurations. It operates in three phases: (1) defining a search space of instances and strategies based on workload, budget, and SLO inputs; (2) predicting performance using an analytical model that captures I/O bottlenecks (PCIe, network, storage), calibrated via linear regression; and (3) identifying the optimal strategy (Scale-up, Multi-pass, or Offloading) and instance combination that maximizes Cost Efficiency (CE) while satisfying strict constraints.

Experimental results demonstrate that LLM-PILOT achieves high prediction accuracy, with a MAPE within 6% in deterministic environments and an $R^2 \geq 0.89$ even in environments subject to queuing delays. In real-world AWS testbed experiments, LLM-PILOT achieved superior CE compared to fixed baseline configurations. Notably, in real-time scenarios, it identified feasible configurations even in regions where existing techniques were classified as *infeasible* due to violations of TPS_{SLO} or latency constraints (TTFT/TBT). Furthermore, across both trace-driven simulations and actual experiments covering diverse instance pools and budget intervals, LLM-PILOT consistently maximized CE among candidates satisfying SLOs, achieving CE improvements of 2.05× compared to High-End strategies and 2.31× compared to state-of-the-art (SOTA) methods.

II. BACKGROUND AND MOTIVATION

A. Architecture of the Transformer

LLMs based on the transformer architecture consist of L stacked layers of the same structure, including self-attention and feed-forward network (FFN) blocks [15]. Each layer calculates query (Q), key (K), and value (V) from the input context S_{in} through D-dimensional hidden vectors, performs self-attention, and then applies the FFN calculations. By repeatedly performing these layer-wise operations, the LLM learns the contextual representation of the input sequence and generates a probability distribution to predict the next token. This LLM inference process is generally divided into a prefill phase and a decode phase [16].

In the prefill phase, the entire input prompt sequence (S_{in}) is processed in parallel to generate the first token. During this process, each layer computes the Q, K, and V matrices for all tokens included in batch size B at once, and the resulting K and V tensors are stored in GPU memory as a KV cache for reuse in subsequent decoding steps. Since the prefill phase is dominated by large-scale matrix multiplication ($B \cdot S_{in} \cdot D$) operations for the entire sequence, the execution time is primarily determined by the GPU's computational throughput, exhibiting *compute-bound* characteristics.

The decode phase sequentially generates the output sequence (S_{out}) according to the autoregressive nature of the model. At each step, while computing Q for the newly generated single token, the K and V states of all previous tokens accumulated up to the current step must be accessed from GPU memory to perform the attention mechanism [17]. Consequently, while the arithmetic intensity required for processing a single token is low, the size of the KV cache that must be transferred at each step increases linearly in proportion to the total sequence length $S = S_{in} + S_{out}$. Due to this data-intensive access pattern, the decode phase exhibits *memory-bound* characteristics, where GPU memory bandwidth and access latency become the performance bottlenecks.

The KV cache continuously accumulates as decoding progresses, and its memory requirement (M_{KV}) can be approximated as follows:

$$M_{\text{KV}} \approx 2 \cdot B \cdot S \cdot L \cdot D \cdot \text{precision} \tag{1}$$

As models that support long contexts of 1M tokens or more become standard, such as Gemini 3 Pro and Sonnet 4.5, there is a clear trend in serving workloads where both batch size (B) and sequence length (S) increase simultaneously [18, 19]. Consequently, the phenomenon where the accumulated KV cache during the decoding stage of an LLM exceeds the physical memory capacity of a single GPU has become a common system constraint [10]. In particular, the adoption of RAG and multimodal workloads rapidly increases the length of input tokens (S_{in}), further intensifying the memory wall problem [20].

B. KV Cache and Attention Offloading for LLM Inference

To alleviate the GPU memory constraints that occur during LLM inference, model parallelism techniques have traditionally been utilized [21, 22]. These methods distribute model parameters and the KV cache across multiple GPUs. However, as the processing of long contexts has become commonplace, managing the massive KV cache solely with GPU memory presents clear limitations in terms of cost-efficiency and scalability.

Therefore, the following two offloading techniques have emerged as essential alternatives: (i) *KV cache offloading*, which stores the KV cache in external memory or storage rather than being limited to internal GPU memory [23, 24, 25], and (ii) *Attention offloading*, which distributes the attention computation load by utilizing heterogeneous GPUs, including both high-performance and low-specification GPUs [26].

*1) **KV cache offloading**:* KV cache offloading (KVO) is a method that stores the KV cache in external memory or storage such as CPU memory or NVMe SSDs instead of GPU memory, which has limited capacity. For each decoding step, KVO dynamically fetches the required KV cache into GPU memory. This technique has the advantage of extending the limits of GPU memory capacity in a cost-effective manner. However, due to the nature of the attention mechanism, the I/O bandwidth between the GPU and external memory becomes a bottleneck as the sequence length increases. This bottleneck is unavoidable as the attention mechanism must reference the KV data for the entire accumulated sequence (S) at every step, thus leading to a sharp decline in inference performance.

*2) **Attention offloading**:* Attention offloading (AO) is a technique that offloads the attention computation into a dedicated machine or auxiliary GPU. This technique takes advantage of the fact that attention operations in the decoding phase have lower arithmetic intensity compared to FFN operations. In this architecture, the main GPU loads the model parameters and is dedicated to high-computation tasks such as QKV projection and FFN, while a physically separated *Attention-Offload Machine (AO Machine)* performs only attention operations. The AO Machine stores the KV cache and shares with the main GPU, performs attention operations by referencing that data, and returns the results to the main GPU. Unlike KVO, this has the advantage of reducing I/O overhead by transmitting only the attention results generated at each decode step, rather than moving the entire KV cache.

In contrast to KVO, which transmits the accumulated cache at every step, AO only sends the current token's activations, offering superior efficiency as sequence length grows. Additionally, since the AO machine requires no model parameters, it can utilize lower-spec, cost-effective devices. However, because communication occurs at every decoding step, performance remains sensitive to inter-node interconnect bandwidth.

C. Challenges of Applying Offloading Strategies for LLM Serving in the Cloud

The problem of serving LLMs that support long contexts in the cloud (long context LLM serving) goes beyond a simple matter of selecting VM instances. In cloud-based LLM serving with offloading, the offloading strategy and the configuration of cost-effective heterogeneous VM clusters represent new and challenging core challenges. In other words, for cost-effective long context LLM serving, one must simultaneously determine the VM combination, the number of instances, bandwidth settings, and the offloading method. This task requires comprehensively considering multi-dimensional factors such as model size, sequence length, and response latency. To achieve this, the following three key challenges must be addressed.

- Complexity in architectural selection
- Inefficiency from rigid resource coupling
- Decision complexity under diverse SLO requirements

First, a decision needs to be made whether to prioritize either the simplicity of the system configuration or cost-efficiency. Users must choose whether to use a single high-end instance like `g6e.xlarge`, which offers high GPU memory capacity and allow for maintaining structural simplicity, or to build a complex but inexpensive structure by applying CPU/disk-based KVO to low-end instances like `g4dn.xlarge`. However, as shown in Table II, even with low-end instances, there exists a trade-off where costs rise sharply whenever instance specifications are upgraded to secure more memory, making it difficult to predict the actual savings. Furthermore, while Spot instances offer significant cost reductions, they introduce preemption risks. Consequently, sophisticated management is essential to leverage their economic benefits without compromising service reliability, particularly for stateless AO nodes.

Second, cloud instances have network performance strongly coupled with CPU and memory specifications, making it impossible to independently scale specific resources (CPU, memory, or network). Taking the `g4dn` family in Table II as an example, to double the network bandwidth (25→50Gbps), an over-provisioning problem occurs where one must unnecessarily pay for eight times the memory (16→128GiB) and incur more than four times the cost. Furthermore, in a distributed configuration, the overall performance is limited by the node with the lowest bandwidth. Thus, to guarantee the efficiency of AO, there is significant economic pressure to configure all instances as high-cost models.

Third, the complexity that SLOs bring to VM selection must also be addressed. As shown in Table I, service scenarios are divided into real-time and batch tasks, each with different latency tolerance ranges. For batch tasks that allow long response times, it is unnecessary to use high-specification instances; conversely, low-specification GPUs do not always necessitate the use of offloading. This is because a resource combination that can satisfy a specific SLO without offloading may be the most cost-effective option. Ultimately, identifying the optimal configuration that complies with SLOs at a minimum cost amidst numerous variables becomes a very challenging decision problem.

In conclusion, offloading-based LLM serving in cloud environments boils down to a high-dimensional optimization problem that must simultaneously satisfy model specifications, infrastructure constraints, and service quality requirements.

However, there is currently a lack of frameworks that systematically support users in building cost-effective VM clusters, forcing them to rely on human intuition or iterative experimentation and to choose sub-optimal configurations.

D. *Existing Approaches and Their Limitations*

Early studies on cloud resource allocation and cost optimization are primarily based on the characteristics of traditional machine learning workloads, thus failing to sufficiently reflect the unique memory occupancy and computation-intensive patterns that occur during LLM inference [11, 13, 27]. Recently, LLM-specific inference optimization techniques have been proposed to fill this gap, but they also expose significant technical blind spots in how they handle the rapidly expanding KV cache.

InferSave [12] is the first study to comprehensively analyze the relationship between cost-efficiency and SLOs by effectively utilizing KVO when serving LLMs in a cloud environment. However, they overlook multi-VM instance environments where AO can be effectively utilized. Furthermore, they only utilize memory as the storage for the KV cache in KVO and do not analyze disk-based KVO implementations.

Studies utilizing heterogeneous GPUs, such as Mélange [14] and SplitWise [28], assume that all KV caches are loaded within the GPU memory. Consequently, these systems exhibit severe scalability bottlenecks when the KV cache exceeds the physical GPU memory capacity in long-context processing or large-batch scenarios.

Additionally, in the current cloud environment, there are no guidelines or systems to automate these complex decision-making processes. Especially in scenarios where offloading is practically essential, such as long context workload scenarios, the method of manually configuring VM clusters and adjusting offloading strategies is unrealistic in terms of time and cost. Therefore, the need for an automated framework that can simultaneously optimize offloading strategies and economical VM cluster configurations is even more evident.

III. Problem Definition

This section formulates the cost-efficiency problem faced by operators running offloading-based LLM inference services in cloud environments. Given this problem, the goal of LLM-Pilot is to determine the optimal serving strategy to maximize cost-efficiency while satisfying a given SLO.

A. *Problem Formulation: Cost Efficiency Maximization*

The optimization problem that LLM-Pilot aims to solve can be defined as follows.

$$\sigma^* = \arg\max_{\sigma \in \Sigma} \mathrm{CE} \tag{2}$$

Here, the serving strategy σ encompasses the offloading method (no offloading, KVO, AO), VM instance configuration, and cluster configuration, while Σ is the set of feasible σ that satisfy the SLO within a given budget and workload. Through this formulation, LLM-Pilot identifies the most cost-effective σ from a complex pool of cloud resources. Table III defines the notations required for the problem definition below.

TABLE II
Various types of instances provided by AWS
(retrieved on November 30, 2025, N. Virginia region).

Name	GPU Type	GPU (#)	On-Demand ($)	Spot ($)	Mem (GiB)	GPU Mem (GiB)	EBS* (Gbps)	Network* (Gbps)
g4dn.xlarge	T4	1	0.526	0.186	16	16	3.5	25
g4dn.4xlarge	T4	1	1.204	0.3189	64	16	4.75	25
g4dn.8xlarge	T4	1	2.176	0.7097	128	16	9.5	50
g4ad.xlarge	V520 Pro	1	0.379	0.104	16	16	3	10
g5.xlarge	A10G	1	0.42	0.132	8	16	3.5	10
g5g.xlarge	T4G	1	0.42	0.13	8	16	4.75	10
g6.xlarge	L4	1	0.805	0.36	16	24	5	10
g6e.xlarge	L40s	1	1.861	0.776	32	48	5	20
g4ad.8xlarge	V520 Pro	2	1.714	0.747	128	32	3	15
g5g.16xlarge	T4G	2	2.744	0.295	128	32	19	25
g4dn.12xlarge	T4	4	3.912	1.097	192	48	9.5	50
g4ad.16xlarge	V520 Pro	4	3.468	1.353	256	64	6	25
g5.12xlarge	A10G	4	5.672	1.808	192	96	16	40
g6.12xlarge	L4	4	4.602	1.718	192	96	20	40
g6e.12xlarge	L40s	4	10.49	3.228	384	192	20	100
g5.48xlarge	A10G	8	16.288	5.69	768	192	768	100
g6e.48xlarge	L40s	8	30.13	8.496	1536	384	60	400
p4d.24xlarge	A100	8	21.957	9.03	1152	320	19	400 (EFA)
p5.48xlarge	H100	8	55.04	14.30	2048	640	80	3200 (EFAv3)

* EBS and Network values represent maximum bandwidth.

B. *Definition of Service Level Objective (SLO)*

a) Inference Latency and Throughput: Before defining the SLO, the actual performance metrics of the system are formulated. For a model with L layers and a batch size of B, the total end-to-end inference time T_{E2E} is approximated as follows:

$$T_{E2E} = L \cdot T_{pre} + L \cdot (S_{out} - 1) \cdot T_{dec} \tag{3}$$

Here, T_{pre} and T_{dec} represent the prefill and decode latency per layer for batch B respectively. From the user's perspective, latency is categorized into Time to First Token (TTFT), the response time for the first token, and Time Between Tokens (TBT), the delay between subsequent token generations. Based on this, the system's throughput metric, Tokens Per Second (TPS), is defined as the total number of tokens processed over the entire inference time. These metrics can be expressed mathematically with the following formulas:

$$\mathrm{TTFT} = L \cdot T_{pre} \tag{4}$$

$$\mathrm{TBT} = \frac{T_{E2E} - TTFT}{S_{out} - 1} = L \cdot T_{dec} \tag{5}$$

$$\mathrm{TPS} = \frac{B \cdot (S_{in} + S_{out})}{T_{E2E}} \tag{6}$$

b) Service-specific SLO: TPS has been widely utilized in recent studies as a key performance metric that simultaneously reflects both throughput and response latency [12, 23]. However, TPS as a single metric has limitations in finely reflecting the varying performance requirements of different service models. Therefore, LLM-Pilot categorizes service types as follows and defines the set of available strategies Σ based on SLO constraints optimized for each environment.

1) **Real-time Service ($M_{service}$ = RT):** In workloads where low-latency responsiveness is essential, such as conversational AI, it is imperative to not only achieve the target throughput (TPS_{SLO}) but also comply with the latency SLOs ($TTFT_{SLO}$, TBT_{SLO}) for individual requests.

$$\sigma \in \Sigma \iff \text{TPS} \geq TPS_{SLO}, \quad \text{s.t.} \begin{cases} \text{TTFT} \leq TTFT_{SLO} \\ \text{TBT} \leq TBT_{SLO} \end{cases} \tag{7}$$

2) **Batch Service ($M_{service}$ = $Batch$):** In batch services aimed at large-scale processing, such as large-scale text analysis, the sole availability condition is securing the minimum throughput required to complete the task within a set time, rather than individual responsiveness.

$$\sigma \in \Sigma \iff \text{TPS} \geq TPS_{SLO} \tag{8}$$

In these definitions, $TTFT_{SLO}$ and TBT_{SLO} are defined by the service requirements determined by the operator's maximum response latency threshold for LLM serving.

C. Cost Efficiency Metric

LLM-PILOT aims not simply to increase TPS, but also to maximize cost-efficiency within the range that achieves the given SLO. In doing so, an important observation is taken into account: over-provisioning that exceeds the SLO does not create additional economic value. For example, in conversational services, once the latency between tokens becomes sufficiently small, additional speed improvements provide negligible benefit on the quality of experience (QoE) [5, 6].

To reflect this, the effective throughput Θ, which takes TPS_{SLO} as its upper bound, is defined as follows: $\Theta = \min(TPS, TPS_{SLO})$. This Θ implies that even if the system provides a high TPS, it is only recognized as effective throughput up to the level of TPS_{SLO}. Since $\sigma \in \Sigma$ already satisfies TPS $\geq TPS_{SLO}$ due to the constraints, Θ is fixed at TPS_{SLO}. Consequently, the problem of maximizing cost-efficiency (CE) is mathematically equivalent to the problem of minimizing the cost C required to process Θ.

The optimization problem can ultimately be formulated as follows:

$$\text{CE} = \frac{\Theta \cdot 3600}{C} \quad \text{[tokens/\$]} \tag{9}$$

$$\sigma^* = \arg\max_{\sigma \in \Sigma} \text{CE}(\sigma) \quad \text{s.t. } C \leq C_{\max} \tag{10}$$

Here, C represents the total hourly cost of all VM instances constituting the serving strategy σ. LLM-PILOT solves this optimization problem to derive the optimal instance configuration and offloading strategy that minimizes costs while complying with the SLO.

IV. DESIGN OF LLM-PILOT

LLM-PILOT consists of an *Analytical Performance Modeler* that mathematically predicts performance by combining hardware specifications with the characteristics of the offloading architecture, and an *Optimization Solver* that searches for the optimal serving strategy based on these predictions.

TABLE III
NOTATION TABLE FOR PROBLEM DEFINITION

Notation	Description
T_{E2E}	Total latency; $T_{E2E} = L \cdot T_{pre} + L \cdot (S_{out} - 1) \cdot T_{dec}$
$T_{\text{pre}}, T_{\text{dec}}$	Latency of prefill and decode stages per layer
TTFT	Time to First Token ($L \cdot T_{pre}$)
TBT	Time Between Tokens ($L \cdot T_{dec}$)
TPS	Actual system throughput (tokens/s)
L	Number of layers in the LLM model
B, S_{in}, S_{out}	Batch size, Input and output sequence lengths
C	Total hourly cost of strategy σ (USD/hour)
$C_{\max}$	Budget constraint (maximum hourly cost)
$M_{service}$	Service model type (RT, Batch)
TPS_{SLO}	Target throughput SLO (tokens/s)
$\text{TTFT}_{\text{SLO}}, \text{TBT}_{\text{SLO}}$	Latency SLO thresholds for RT service
Θ	Effective throughput; $\min(\text{TPS}, \text{TPS}_{\text{SLO}})$
σ, Σ	Serving strategy and its set

A. Architectural Workflow

LLM-PILOT's overall workflow can be divided into three steps.

1) **Input Specification:** The system receives the workload $\mathcal{W} = (B, S_{in}, S_{out})$, the SLO requirements $\mathcal{R} = (M_{service}, TPS_{SLO}, TTFT_{SLO}, TBT_{SLO})$, and the hourly budget constraint $C_{\max}$ as input from the user. Based on these inputs, the total search space Σ is defined as the set of all possible combinations of instance configurations and offloading strategies (no offload, KVO, AO).
2) **Analytical Performance Modeler:** For each serving strategy $\sigma \in \Sigma$, performance metrics (TPS, TTFT, TBT) are mathematically predicted by combining hardware specifications with the characteristics of the offloading architecture. The predicted values are calibrated using linear regression based on offline profiling data to minimize the error compared to actual measured performance.
3) **Optimization Solver:** Based on the calibrated prediction model, the optimal serving strategy σ^* is derived. First, the search space is reduced by pre-filtering candidates expected to exceed memory capacity or violate SLOs. Subsequently, within the set of feasible strategies Σ_{feas} that satisfy the SLO within the given budget, the σ^* with the highest cost-efficiency metric CE is selected.

Table IV shows the key variables and notations used in LLM-PILOT.

B. Analytical Performance Modeling

The *Analytical Performance Modeler* is employed in LLM-PILOT to efficiently evaluate the vast search space. The core purpose of the model is to abstract the physical costs of LLM inference into three hardware-independent functions as shown in Table V: Compute ($\mathcal{F}$), Storage ($\mathcal{M}$), and Transfer ($\mathcal{D}$). Based on this abstraction, the goal is to identify bottlenecks by combining these functions with the resource bandwidth of each architecture.

The formula $T_{\text{E2E}} = L \cdot T_{\text{pre}} + L \cdot (S_{out} - 1) \cdot T_{\text{dec}}$ defined in §III applies equally to all serving strategies. Each offloading strategy differs only in the calculation methods of T_{pre} and

TABLE IV
KEY NOTATIONS FOR MODELING PARAMETERS.

Sym.	Description	Sym.	Description
	Input Specification		
$\mathcal{W}$	Workload (B, S_{in}, S_{out})	$C_{\max}$	Budget constraint
$\mathcal{R}$	SLO requirements		
	Model Parameters		
L	# of layers	h	Hidden dimension
b_w	Bytes per element	S	Total seq. len $(S_{in} + S_{out})$
	Hardware Resources (from Instance Pool $\mathcal{I}$)		
C_{gpu}	GPU compute (FLOPS)	BW_{mem}	Memory bandwidth
BW_{net}	Network bandwidth	BW_{io}	PCIe/Disk bandwidth
	Decision Variables		
α	Offload ratio	k	# of micro-batches
N_{ao}	# of AO workers	σ	Serving strategy

T_{dec} per layer, which are defined below, with the batch size B included in Table V.

*1) **KV Cache offloading (KVO)**:* KVO overcomes GPU memory limits by storing a portion (ratio α) of the KV cache in an external tier (CPU memory or storage). As a result, the bandwidth of the external interface, BW_{io}, acts as a new bottleneck.

Prefill Phase: A fraction α of the KV cache generated in each layer is asynchronously transferred to external memory or storage. Since computation and transfer are performed in parallel, the slower of the two determines the latency.

$$T_{\text{pre}}^{\text{KVO}} = \max\left(\frac{\mathcal{F}(B, S_{in})}{C_{\text{gpu}}}, \frac{\alpha \cdot \mathcal{D}_{\text{kvo}}(B, S_{in})}{\text{BW}_{\text{io}}}\right) \tag{11}$$

Decode Phase. At each step t, a ratio α of the KV cache is fetched from external memory or storage, while a fraction $(1-\alpha)$ is read from the GPU memory. GPU memory access and external I/O are performed in parallel, and the attention operation is executed only after all KV cache loading is complete. Therefore, the latency is modeled as follows:

$$T_{\text{dec}}^{\text{KVO}} = \frac{\mathcal{F}(B, l_t)}{C_{\text{gpu}}} + \max\left(\frac{(1-\alpha)\cdot \mathcal{M}(B, l_t)}{\text{BW}_{\text{mem}}}, \frac{\alpha \cdot \mathcal{D}_{\text{kvo}}(B, l_t)}{\text{BW}_{\text{io}}}\right) \tag{12}$$

Here, $l_t = S_{in} + t$ represents the cumulative sequence length at decoding step t. Since the external I/O term increases in proportion to l_t ($\mathcal{O}(B \cdot l_t)$), the performance of KVO degrades significantly for long sequences.

*2) **Attention offloading (AO)**:* AO is a strategy where the main GPU handles compute-intensive FFN and projection tasks exclusively, while the memory-intensive attention operation is shared between the main GPU and the AO Machine. The offloading ratio α represents the proportion of queries from the total batch B allocated to the AO Machine to perform attention.

Prefill Phase: After the main GPU generates the KV cache for the input sequence, it transfers a ratio of α to the AO Machine. Similar to KVO, this process is limited by the network bandwidth (BW_{net}).

TABLE V
LAYER-WISE COMPLEXITY ABSTRACTION FOR TRANSFORMER INFERENCE. KVO AND AO EXHIBIT DISTINCT TRANSFER CHARACTERISTICS DURING DECODING.

Metric	Component	Prefill	Decode (step t)
Compute ($\mathcal{F}$) (per layer)	FLOPs **Dominant Ops*	$B \cdot S_{in}(24h^2 + 4S_{in}h)$ *MatMul (Proj, FFN) + Attn*	$B(24h^2 + 4l_t h)$
Storage ($\mathcal{M}$) (Bytes)	KV Capacity **State Size*	$2B \cdot S_{in} \cdot h \cdot b_w$ *Accumulated KV Cache*	$2B \cdot l_t \cdot h \cdot b_w$
Transfer ($\mathcal{D}$) (Bytes)	**I/O (KVO)** **Bottleneck*	$2B \cdot S_{in} \cdot h \cdot b_w$ (Swap-out)	$\mathbf{2B \cdot l_t \cdot h \cdot b_w}$ (Swap-in: $\mathcal{O}(l_t)$)
	Comm (AO) **Bottleneck*	$4B \cdot S_{in} \cdot h \cdot b_w$ (Send KV to AO)	$\mathbf{4B \cdot h \cdot b_w}$ (Const: $\mathcal{O}(1)$)

* $l_t = S_{in} + t$: cumulative sequence length at decode step t.

$$T_{\text{pre}}^{\text{AO}} = \max\left(\frac{\mathcal{F}(B, S_{in})}{C_{\text{gpu}}^{\text{Main}}}, \frac{\alpha \cdot \mathcal{D}_{\text{AO}}(B, S_{in})}{\text{BW}_{\text{net}}}\right) \tag{13}$$

Decode Phase: The main GPU performs attention calculations for $(1-\alpha)B$ queries, while simultaneously, the AO Machine processes αB queries. The two paths execute in parallel, and after the FFN operation, the critical path determines the total latency.

$$T_{\text{dec}}^{\text{AO}} = \frac{\mathcal{F}_{\text{ffn}}(B)}{C_{\text{gpu}}^{\text{Main}}} + \max\left(T_{\text{local}}, T_{\text{remote}}\right) \tag{14}$$

Here, T_{local} is $\frac{\mathcal{F}_{\text{attn}}((1-\alpha)B, l_t)}{C_{\text{gpu}}^{\text{H}}}$, and T_{remote} is $\frac{\mathcal{F}_{\text{attn}}(\alpha B, l_t)}{C_{\text{gpu}}^{\text{AO}}} + \frac{\mathcal{D}_{\text{ao}}(B)}{\text{BW}_{\text{net}}}$.

As shown in Table V, the transfer volume of AO, $\mathcal{D}_{\text{ao}}$, is $\mathcal{O}(B)$ regardless of the sequence length. Therefore, unlike KVO, the transfer overhead remains constant even for long sequences.

*3) **Advanced AO Modeling: Parallel & Shared**:* To overcome the limitations of the basic $1:1$ AO configuration and maximize cost-efficiency, two extended topologies are included in the modeling.

Parallel AO (K-Pairs). In this topology, K 1:1 AO pairs are deployed in parallel, processing the total batch B divided into K equal parts. While the cost increases by a factor of K, the latency is reduced, enabling the fulfillment of strict SLO constraints.

$$T_{\text{dec}}^{\text{Parallel}} = T_{\text{dec}}^{\text{AO}}\left(\frac{B}{K}, l_t\right) \tag{15}$$

This strategy is viable for high-performance services that must satisfy strict SLO constraints, at the expense of cost-efficiency.

Shared AO (N:1). In this topology, N main GPUs share a single AO Machine to increase resource utilization. As multiple requests converge, queuing delay occurs; since the computation load is deterministic, it is modeled as an M/D/1 queue [29]. The total decoding time is calculated by adding the average waiting time (W_q) to the base latency ($T_{\text{dec}}^{\text{AO}}$) of the non-congested case.

$$T_{\text{dec}}^{\text{Shared}} = T_{\text{dec}}^{\text{AO}} + \underbrace{\frac{\rho}{2\mu(1-\rho)}}_{W_q} \tag{16}$$

Here, μ^{-1} is the service time of the AO Machine, and $\rho = \lambda/\mu < 1$ represents the system load. As ρ approaches 1, the waiting time W_q increases sharply, increasing the risk of SLO violations.

*4) **Single-Node Inference with Multi-Pass**:* When performing inference on a single high-performance GPU without offloading, the GPU memory may be unable to accommodate the KV cache for the entire batch B. In this case, the batch can be divided into k micro-batches and processed sequentially. Since each micro-batch executes independently and KV cache reuse is impossible, the total latency increases by k times that of a single pass.

$$T_{\text{E2E}}^{\text{MP}} = k \cdot T_{\text{E2E}}\left(\frac{B}{k}\right) \tag{17}$$

While this strategy can resolve memory constraints while avoiding the complexity of offloading configurations, the linear decrease in throughput is a key trade-off.

C. Calibration via Linear Regression

Since the analytical model assumes ideal hardware specifications, discrepancies arise between the model and actual system performance. In actual LLM inference, phenomena such as degraded GPU utilization occurs during attention operation due to irregular memory access patterns, or effective bandwidth decreases due to resource contention during offloading. According to previous research, such systemic inefficiencies exhibit a linear correlation with the number of input tokens or the batch size [12]. Based on these observations, the model's prediction error is calibrated using linear regression.

$$T_{\text{real}} = \beta_1 \cdot T_{\text{pred}} + \beta_0 \tag{18}$$

Here, the slope β_1 represents the inefficiency factor of the actual hardware compared to the theoretical performance, while the intercept β_0 compensates for fixed latencies such as kernel launch overhead. Since this profiling incurs only a one-time offline cost for each hardware configuration, it does not affect runtime performance.

D. Optimization Solver

LLM-PILOT utilizes a calibrated performance model as the *Optimization Solver* to derive the most cost-efficient serving strategy σ^* within the entire search space Σ. To maximize search efficiency, a two-stage process consisting of *Feasibility Filtering* and *Cost-Efficiency Optimization* is performed.

1) **Feasibility Filtering:** Strategies that are physically infeasible or fail to satisfy the SLO constraints are excluded from the search space.
   - **Resource Constraints:** Cases where the total available memory (GPU, host RAM) cannot accommodate the model parameters and KV cache requirements.
   - **Performance Constraints:** Cases where the predicted latency or throughput fails to satisfy the SLO requirements $\mathcal{R}$.

   This process reduces the entire search space Σ to a set of valid strategies $\Sigma_{\text{feas}} \subset \Sigma$.
2) **Cost-Efficiency Optimization:** For each strategy σ in Σ_{feas}, the cost-efficiency metric CE(σ) is calculated. Here, the effective throughput $\Theta = \min(\text{TPS}, \text{TPS}_{SLO})$ is used to penalize over-provisioning that exceeds the target. Finally, by selecting $\sigma^* = \arg\max_{\sigma \in \Sigma_{\text{feas}}} \text{CE}(\sigma)$, the optimal combination of instances and offloading strategies that minimizes cost under the given SLO is finalized.

E. Implementation

We implemented the prototype of LLM-PILOT using Python (v3.12.3). The system is built upon FlexLLMGen (v0.1.7), an LLM inference framework that natively supports KV cache offloading. Our implementation comprises approximately 1.1k lines of Python code dedicated for the inter-machine communication module. The optimization solver was implemented in Python, alongside a trace-driven simulation environment developed to validate its decision-making capabilities. We used SciPy (v1.15.3) for latency calibration tasks. For tensor operations and GPU execution, we employed NumPy (v2.3.3) and PyTorch (v2.7.0+cu128) compatible with CUDA 12.8. To enable distributed execution without modifying the source code of the original inference engine, we adopted a communication protocol based on memory-mapped files over tmpfs-backed NFS.

V. EVALUATION

In this section, we verify the efficacy and robustness of LLM-PILOT through the following three research questions (RQs).

- **(RQ1) Model Accuracy:** Does the proposed performance model accurately predict actual latency under various hardware bottleneck conditions?
- **(RQ2) Economic Validity:** Does the framework achieve substantial improvements in cost-efficiency (CE, tokens/\$) compared to existing baselines under diverse cost/resource constraints and price fluctuations?
- **(RQ3) Decision Dynamics:** Does the solver dynamically switch to the optimal strategy in response to changes in workload characteristics and SLO constraints?

A. Experimental Setup

Our evaluation consists of (i) trace-driven simulations and (ii) AWS Testbed experiments for real-world validation to cover combinatorial configurations infeasible to explore exhaustively on real instances.

A1. Common Evaluation Protocol:

- **Metrics:** Cost efficiency is defined as Cost Efficiency (CE, tokens/\$), representing the number of tokens processed per unit cost. Throughput is measured in Tokens Per Second (TPS, tokens/s). For real-time services, to ensure response quality, we incorporate latency metrics: the average Time To First Token (TTFT, ms) and the P95 value of Time Between Tokens (TBT, ms) across queries in a batch. For the sake of brevity, we omit these units in the following text and tables.

TABLE VI
DETAILS OF THE SIMULATED INSTANCE TYPES USED IN TRACE-DRIVEN SIMULATIONS.

Name	VRAM (GiB)	RAM (GiB)	EBS BW (Gbps)	Net. BW (Gbps)	On-Demand ($/hr)	Spot ($/hr)	Category (*Base Instance*)
Sim-A	16	16	3.0	10	0.379	0.104	*Entry (g4ad)*
Sim-B	16	8	4.75	10	0.420	0.132	*Low Cost (g5g)*
Sim-C	16	16	3.5	25	0.526	0.186	*Baseline (g4dn)*
Sim-D	24	16	5.0	10	0.805	0.360	*Cost-Opt (g6)*
Sim-E	24	24	3.5	10	1.006	0.403	*Standard (g5)*
Sim-F	24	64	4.75	25	1.624	0.590	*High RAM (g5)*
Sim-G	48	32	5.0	20	1.861	0.370	*High Perf (g6e)*
Sim-H	16	32	9.5	50	2.176	0.710	*High Net (g4dn)*
Sim-I	48	64	10.0	25	4.528	1.506	*Over-spec (g6e)*

- **Feasibility:** Experimental configurations are classified as *feasible* and considered for cost-efficiency calculation only if they satisfy the target throughput (TPS_{SLO}).
  - **Real-time Service:** Both throughput and latency constraints must be satisfied ($\text{TPS} \geq 100 \wedge \text{TTFT} \leq 200 \wedge \text{TBT} \leq 50$). Any violation results in the configuration not being considered.
  - **Batch Service:** Configurations are classified as *feasible* if they satisfy only the throughput condition ($\text{TPS} \geq 10$).

A2. Datasets and Workloads:
We utilized the `OPT-6.7B` model for the experiments and sampled prompts from the public conversation and instruction datasets, ShareGPT [30] and Alpaca [31]. To observe the variations in the solver's selection relative to system bottlenecks, we employ three representative workload profiles defined by input/output length combinations (S_{in}, S_{out}):

- *Generative* (S_{in} = 128, S_{out} = 1024)
- *Summarization* (S_{in} = 1024, S_{out} = 128)
- *Balanced* (S_{in} = 512, S_{out} = 512)

A3. Simulation Setup (Sensitivity Analysis):
We employ simulations to analyze whether the LLM-PILOT solver effectively selects cost-efficient configurations from a candidate pool exhibiting diverse hardware constraints (VRAM, RAM, EBS bandwidth, and network bandwidth) and pricing models (On-Demand and Spot). Table VI summarizes the instance pool utilized in the simulations. We fix the simulation workload to the *Balanced* profile (S_{in} = 512, S_{out} = 512) to incorporate contributions from both the prefill and decode phases simultaneously. Furthermore, we analyze the selection results and Cost Efficiency(CE) across different budget intervals by incrementally increasing the budget constraint $C_{\max}$ from \$0.5 to \$5.0.

A4. AWS Testbed and Baselines:
To validate the framework in a real-world cloud environment, we conducted experiments in the AWS us-east-1 region. The availability of GPU instances on AWS is subject to fluctuations based on region, time, account-specific quotas (vCPU), and availability policies. Consequently, allowing unrestricted exploration of instance combinations for each comparison strategy during the experimental phase risks skewing results due to instance selection variance rather than intrinsic strategic differences, leading to potential cherry-picking. To mitigate

TABLE VII
AWS EC2 INSTANCE SPECIFICATIONS (US-EAST-1 REGION).

Instance	GPU	Mem. (GiB) *VRAM / RAM*	NVMe *(GB)*	BW (Gbps) *Net / EBS*	Price ($/hr) *On-Dem / Spot*
`g5.xlarge`	A10G	24 / 16	250	10 / 3.5	1.006 / 0.403
`g5.4xlarge`	A10G	24 / 64	600	25 / 4.8	1.624 / 0.590
`g5.8xlarge`	A10G	24 / 128	900	25 / 16	2.448 / 0.734
`g6e.2xlarge`	L40S	48 / 64	450	20 / 5.0	2.242 / 0.592
`g4dn.xlarge`	T4	16 / 16	125	25 / 3.5	0.526 / 0.186
`g4dn.4xlarge`	T4	16 / 64	225	25 / 4.8	1.204 / 0.319

this, we define a fixed instance pool as shown in Table VII, and all baselines are configured within this identical pool while adhering to predefined rules, as follows:

- **High-End:** Inference is performed on a single `g6e.2xlarge` (L40S, 48GB VRAM) node. If the GPU memory capacity is exceeded, the MultiPass strategy is applied to split and execute requests; this is denoted as **High-End(MP)**.
- **AO Cluster:** Attention Offloading (AO) is applied in a 2-node cluster consisting of a `g5.xlarge` (Main) and a `g4dn.4xlarge` (AO-Machine).
- **KVO (Disk):** Storage KVO is applied on a single `g4dn.xlarge` node to offload the KV cache to storage (EBS).
- **InferSave:** We follow the configuration of InferSave [12], the state-of-the-art method utilizing host memory-based KVO. The base instance is `g5.4xlarge`; if memory capacity is exceeded, a vertical scaling rule to `g5.8xlarge` is applied, denoted as **InferSave+**.

B. (RQ1) Model Validation with Calibration via Linear Regression

We validate the prediction accuracy of the proposed Analytical Performance Model under diverse hardware bottleneck scenarios, using the baselines and specifications defined in Table VII. To stress-test the model, we configured a 100% offloading scenario (α = 1.0), where the entire KV cache is transferred externally from the GPU, maximizing the impact of network and I/O bottlenecks. Table VIII presents snapshot results at $B = 32$ and aggregate metrics across $B = 1{\sim}32$.

- **High-End Configuration (L40S Single GPU):** In the compute-bound single-node environment, the model achieved very high prediction accuracy with $R^2 = 0.99$ and MAPE $\leq$ 2.3%. Maintaining consistent accuracy across all three workloads demonstrates that the $\mathcal{F}/C_{\text{gpu}}$-based computation modeling precisely reflects actual hardware behavior.
- **AO Cluster Configuration (1:1):** Although the error slightly increased compared to the high-end configuration due to added network communication overhead, the model still maintained robust accuracy with $R^2 \geq 0.95$ and MAPE $\leq$ 6.2%. Notably, the generative workload (MAPE=4.5%) exhibited higher accuracy than Summarization (MAPE=6.2%) because communication patterns are more regular during the long decoding process.
- **AO Cluster Configuration (Shared, N:1):** In the $N : 1$ topology where requests from multiple hosts converge on a single offloader, latency variance increased due to queuing

TABLE VIII
MODEL PREDICTION ACCURACY UNDER 100% OFFLOADING (α = 1.0). SNAPSHOT RESULTS AT B = 32 AND AGGREGATE METRICS (R^2, MAPE) ACROSS B = 1∼32.

Config.	Workload *(In/Out)*	Snapshot (B = 32)			Metric (B = 1∼32)	
		Real *(sec)*	**Pred** *(sec)*	**Err** *(%)*	R^2 *(0-1)*	**MAPE** *(%)*
High-End	Gen (128/1024)	39.4	38.5	2.3	**0.99**	**2.1%**
	Bal (512/512)	20.8	20.3	2.4	**0.99**	**2.2%**
	Sum (1024/128)	8.2	8.0	2.5	**0.99**	**2.3%**
AO (1:1)	Gen (128/1024)	68.5	65.2	4.8	**0.97**	**4.5%**
	Bal (512/512)	36.4	34.5	5.2	**0.96**	**5.1%**
	Sum (1024/128)	11.8	10.9	7.6	**0.95**	**6.2%**
AO (N:1)	Gen (128/1024)	78.6	73.1	7.0	**0.95**	**5.8%**
	Bal (512/512)	41.2	38.5	6.5	**0.94**	**6.4%**
	Sum (1024/128)	13.5	12.4	8.1	**0.93**	**7.1%**
KVO (Disk)	Gen (128/1024)	**368.5**	335.2	9.0	**0.89**	**11.2%**
	Bal (512/512)	189.2	172.5	8.8	**0.91**	**9.5%**
	Sum (1024/128)	49.8	45.6	8.4	**0.94**	**8.2%**

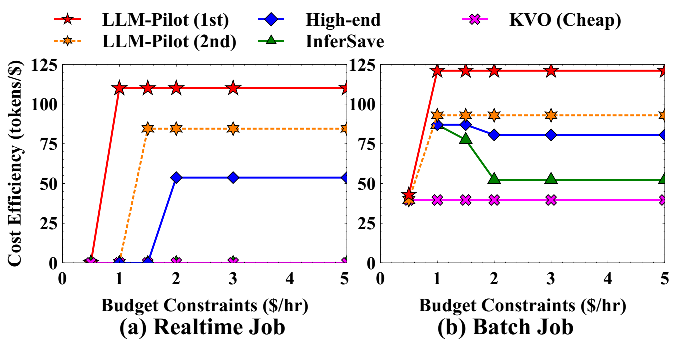


Fig. 1. Cost efficiency (CE) comparison under varying budget constraints. (a) Real-time scenario (TPS_{SLO}=100). (b) Batch scenario (TPS_{SLO}=10).

delay (R^2 = 0.93∼0.95). However, by limiting MAPE within 7.1% through modeling based on M/D/1 queue theory, we confirmed the validity of the proposed model even in complex topologies.

- **Storage KVO Configuration:** Due to the variability of disk I/O latency and irregular KV cache access patterns, this configuration recorded the highest prediction error among the baselines (MAPE=11.2%). However, the model accurately captured the extreme latency scale reaching 368 seconds in the generative workload (R^2 = 0.89), providing sufficient reliability for SLO feasibility determination.

In summary, the proposed model achieves high accuracy with MAPE $\leq$ 6% in deterministic environments (High-End, AO 1:1) and provides sufficient reliability for feasibility determination even in environments involving stochastic factors (AO Shared, KVO).

C. (RQ2) Validation in Simulation

We validate the economic validity of LLM-PILOT using the trace-driven simulation described in §V-A, comparing CE (tokens/$) across strategies under varying budgets. Fig. 1 summarizes the results. Moreover, Table IX highlights the results of the validation in simulation test.

*1) **Scenario (a): Real-Time Service:*** Under strict SLOs, LLM-PILOT identified the AO Cluster (Sim-D + Sim-A, $0.46/hr) as the optimal *1st-pick*, achieving CE=110. In contrast, single-node baselines were severely limited as shown in Fig. 1(a).

*a) **$1.0–$1.5/hr: Low-Budget Region:*** LLM-PILOT becomes the only feasible solution, achieving CE=110 by leveraging inter-node network bandwidth for KV cache offloading. All other baselines remain infeasible: the cheapest High-End option (Sim-G at on-demand) exceeds the budget, while InferSave and KVO cannot meet the SLO at any price due to PCIe and disk I/O bottlenecks, respectively.

*b) **$2.0+/hr: High-Budget Region:*** Above $2.0/hr, the High-End baseline (Sim-G) becomes budget-feasible and achieves CE=53.7. However, LLM-PILOT retains its AO Cluster configuration and delivers 2.05× higher CE (110 vs. 53.7), corresponding to a 51% cost reduction for equivalent throughput.

*c) **Fallback Strategy Effectiveness:*** To mitigate Spot preemption risk, LLM-PILOT provides an On-Demand *2nd-pick* (Sim-D + Sim-A, CE=84), which still achieves 57% higher CE than the High-End baseline.

TABLE IX
QUANTITATIVE CE COMPARISON UNDER TPS_{SLO}=100 (REAL-TIME) AND TPS_{SLO}=10 (BATCH).

Scenario	Budget	LLM-Pilot	InferSave	High-End	Improvement
Real-time	$1.0–1.5	110.0	Infeasible	Infeasible	*Only feasible*
Real-time	$2.0+	110.0	Infeasible	53.7	**2.05**× vs High-End
Batch	$1.0	121.0	87.0	87.0	**1.39**× vs Baselines
Batch	$2.0+	121.0	52.3	80.6	**2.31**× vs InferSave

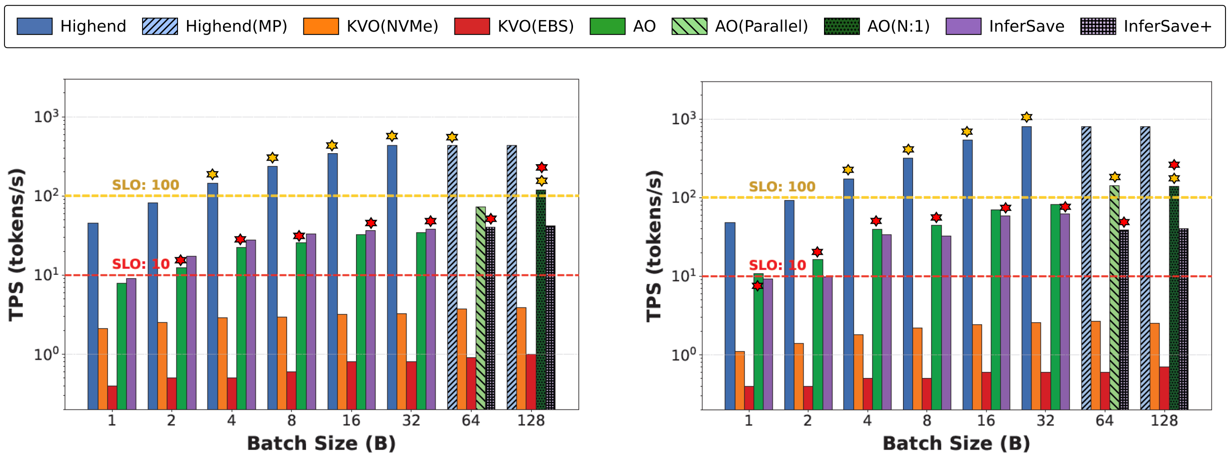
*2) **Scenario (b): Batch Service:*** With relaxed constraints (TPS_{SLO} = 10), more strategies become feasible, yet LLM-PILOT still achieves dominant CE. The solver selected the same Spot-based AO Cluster (Sim-D + Sim-A) as the *1st-pick* (CE=121), while the relaxed SLO now allows InferSave, KVO (Cheap), and High-End to become feasible, as shown in Fig. 1(b).

*a) **$1.0–$1.5/hr: Low-Budget Region:*** In this range, InferSave (Sim-E, CE=87), High-End (Sim-G, CE=87) and KVO Cheap (Sim-A, CE=42) become feasible. However, LLM-PILOT (CE=121) already outperforms InferSave and High-End by 1.39× and KVO by 2.88×, reflecting the same PCIe and I/O bottlenecks identified in Scenario (a).

*b) **$2.0+/hr: High-Budget Region:*** As the budget increases, InferSave applies vertical scaling from Sim-E to Sim-F ($0.40→$0.59/hr), but exhibits sub-linear scaling with CE degrading by 40% (87→52.3) as cost grows faster than performance. High-End also decreases slightly to CE=80.6 due to the transition to higher-cost pricing. LLM-PILOT maintains CE=121, delivering 2.31× higher efficiency than InferSave and 1.50× over High-End.

*c) **Fallback Strategy Effectiveness:*** The On-Demand *2nd-pick* achieves CE=92.9, still 78% higher than InferSave and still 15.3% higher than High-End in the $2.0+/hr range.

In conclusion, LLM-PILOT (1) provides the sole feasible solution where baselines fail in Real-time scenarios, (2) delivers 1.39∼2.31× CE improvements in Batch scenarios, and (3) offers robust fallback strategies.



(a) Summarization (1024/128) (b) Generative (128/1024)

Fig. 2. Throughput (TPS) comparison across serving strategies on AWS Testbed. SLO thresholds (10 and 100 tokens/s) are marked as dashed lines. Hatched bars indicate MultiPass or Parallel variants. Star (Red/Yellow) means that each bar is the most cost efficient.

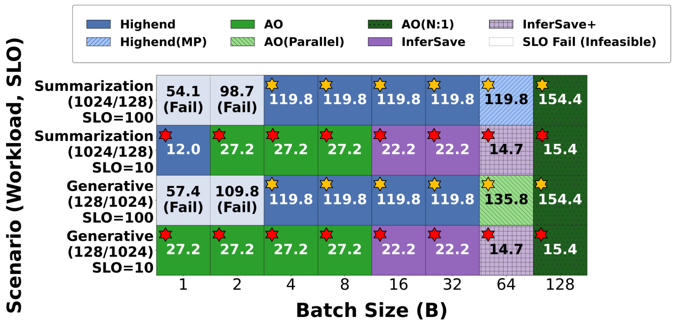


Fig. 3. Decision dynamics map predicted by LLM-PILOT and validated on AWS Testbed, showing the optimal strategy for each combinations. Cell values indicate CE (tokens/$). Star (Red/Yellow) means that each cell is also selected in the AWS Testbed.

D. *(RQ3) Decision Dynamics on AWS Testbed*

This experiment validates whether the solver's predicted optimal aligns with the ground-truth optimal measured on AWS. Fig. 2 presents the measured throughput (TPS), and Fig. 3 illustrates the decision dynamics map, which shows the selected strategy and CE (tokens/$) for each workload–SLO–batch combination (B = 1∼128). Stars indicate cells where prediction matched the ground-truth Oracle on AWS. Note that absolute CE values differ in scale from §V-C due to different workload profiles and instance pools.

Prediction Accuracy. LLM-PILOT identified the optimal strategy consistent with the Oracle across the entire decision space. This confirms that the performance model reliably ranks strategies under diverse bottleneck conditions without exhaustive online profiling.

Real-Time (TPS$_{SLO}$ = 100): Phase Transitions. At $B \leq 2$, no strategy satisfies the joint throughput-latency constraints. From $B = 4$, AO Cluster dominates the mid-range (B = 4∼32) via inter-node offloading. At B = 64, a workload-dependent divergence emerges: for Summarization, the High-End strategy hits the VRAM wall and transitions to High-End(MP); for Generative, AO(Parallel) is selected instead, as parallelized offloading scales more cost-effectively under long decode sequences. At B = 128, AO(N:1) becomes dominant across both workloads, as the shared-offloader topology amortizes communication overhead at large batch sizes.

Batch (TPS$_{SLO}$ = 10): Fine-Grained Differentiation. At B = 1, the choice diverges by workload: High-End for Summarization and AO for Generative. At B = 2∼8, AO dominates as inter-node offloading efficiently handles moderate KV cache volumes. At B = 16∼32, the solver transitions to InferSave, where host-memory offloading via PCIe provides more stable throughput as KV cache size grows. At B = 64, InferSave+ is selected to accommodate the increased memory demand via vertical scaling. At B = 128, the solver switches to AO(N:1), where the shared-offloader topology enables pipelined transfers from multiple hosts, recovering network efficiency at large batch sizes.

In conclusion, LLM-PILOT achieves full Oracle alignment across all 32 combinations, correctly identifying infeasibility boundaries, VRAM-induced transitions, and workload-dependent strategy shifts.

VI. CONCLUSION

In this paper, we propose LLM-PILOT, a framework that automatically derives VM cluster configurations for cost-efficient LLM inference in cloud environments. Given a workload and its SLOs, LLM-PILOT constructs a search space of candidate instances and strategies, then uses a performance-model-based analysis to select the configuration that maximizes cost efficiency while satisfying all constraints by jointly accounting for offloading strategies (KVO and AO). By jointly optimizing offloading decisions and instance selection, LLM-PILOT offers a systematic approach to balancing cost and performance across diverse cloud LLM serving workloads.

ACKNOWLEDGEMENT

This work was partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2024-00416666, RS-2025-00564249).

References

[1] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, and et. al., "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2024.

[2] OpenAI, "OpenAI Platform API." [Online]. Available: https://platform.openai.com/docs/pricing. Accessed: Dec. 11, 2025.

[3] OpenAI, "OpenAI Pricing." [Online]. Available: https://openai.com/api/pricing. Accessed: May 17, 2025.

[4] Anthropic, "Message Batches API: Run jobs up to 50% cheaper." [Online]. Available: https://www.anthropic.com/news/message-batches-api. Accessed: Dec. 30, 2025.

[5] K. Rayner, E. R. Schotter, M. E. J. Masson, M. C. Potter, and R. Treiman, "So much to read, so little time: how do we read, and can speed reading help?," *Psychological Science in the Public Interest*, vol. 17, no. 1, pp. 4–34, 2016.

[6] H. Li, Y. Liu, Y. Cheng, S. Ray, K. Du, and J. Jiang, "Eloquent: A more robust transmission scheme for LLM token streaming," in *Proceedings of the 2024 ACM SIGCOMM Workshop on Networks for AI Computing (NAIC)*, pp. 34–40, 2024.

[7] Amazon Web Services, "Amazon EC2 Instance Types." https://aws.amazon.com/ec2/instance-types/. [Online]. Accessed: Dec. 11, 2025.

[8] Y. Ding, L. L. Zhang, C. Zhang, Y. Xu, N. Shang, J. Xu, F. Yang, and M. Yang, "Longrope: Extending llm context window beyond 2 million tokens," *arXiv preprint arXiv:2402.13753*, 2024.

[9] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, and et al., "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[10] K. Chu, Z. Shen, S.-R. Cheng, D. Xiang, Z. Liu, and W. Zhang, "MCaM: Efficient LLM inference with multi-tier KV cache management," in *Proceedings of the 45th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 571–581, 2025.

[11] Y. Kim, K. Kim, Y. Cho, J. Kim, A. Khan, K.-D. Kang, B. S. An, M.-H. Cha, H.-Y. Kim, and Y. Kim, "DeepVM: Integrating spot and on-demand VMs for cost-efficient deep learning clusters in the cloud," in *Proceedings of the 24th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 227–235, 2024.

[12] K. Kim, J. Kim, H. Chung, M.-H. Cha, H.-Y. Kim, and Y. Kim, "Cost-efficient VM selection for cloud-based LLM inference with KV cache offloading," in *Proceedings of the 18th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 175–185, 2025.

[13] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: Agile ML elasticity through tiered reliability in dynamic resource markets," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pp. 589–604, 2017.

[14] T. Griggs, X. Liu, J. Yu, D. Kim, W.-L. Chiang, A. Cheung, and I. Stoica, "Mélange: Cost efficient large language model serving by exploiting gpu heterogeneity," *arXiv preprint arXiv:2404.14527*, 2024.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, pp. 5998–6008, 2017.

[16] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve," in *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 117–134, 2024.

[17] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, 2023.

[18] Google, "Gemini 3.0 pro Context Window Size." [Online]. Available: https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/3-pro. Accessed: Dec. 17, 2025.

[19] A. Yang, B. Yu, C. Li, D. Liu, F. Huang, H. Huang, J. Jiang, J. Tu, J. Zhang, J. Zhou, and et. al., "Qwen2.5-1m technical report," *arXiv preprint arXiv:2501.15383*, 2025.

[20] Q. Leng, J. Portes, S. Havens, M. Zaharia, and M. Carbin, "Long context rag performance of large language models," *arXiv preprint arXiv:2411.03538*, 2024.

[21] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS)*, pp. 103–112, Curran Associates Inc., 2019.

[22] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[23] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "FlexGen: High-throughput generative inference of large language models with a single GPU," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*, JMLR.org, 2023.

[24] Y. Xiong, H. Wu, C. Shao, Z. Wang, R. Zhang, Y. Guo, J. Zhao, K. Zhang, and Z. Pan, "LayerKV: Optimizing large language model serving with layer-wise KV cache management," 2024.

[25] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "DeepSpeed-Inference: Enabling efficient inference of transformer models at unprecedented scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, IEEE Press, 2022.

[26] X. Pan, E. Li, Q. Li, S. Liang, Y. Shan, K. Zhou, Y. Luo, X. Wang, and J. Zhang, "Instinfer: In-storage attention offloading for cost-effective long-context llm inference," *arXiv preprint arXiv:2409.04992*, 2024.

[27] G. Fragiadakis, V. Liagkou, E. Filiopoulou, D. Fragkakis, C. Michalakelis, and M. Nikolaidou, "Cloud services cost comparison: A clustering analysis framework," *Computing*, vol. 105, no. 10, pp. 2061–2088, 2023.

[28] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: efficient generative llm inference using phase splitting," in *Proc. of the 51st ACM/IEEE Annu. Int. Symp. on Computer Architecture (ISCA)*, pp. 118–132, IEEE, 2024.

[29] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. USA: Wiley-Interscience, 1975.

[30] ShareGPT, "ShareGPT: Share your wildest ChatGPT conversations." [Online]. Available: https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered. Accessed: Dec. 22, 2025.

[31] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford Alpaca: An Instruction-following LLaMA Model." [Online]. Available: https://huggingface.co/datasets/tatsu-lab/alpaca. Accessed: 2025-12-22.