Rock: Serving Multimodal Models in Cloud with Heterogeneous-Aware Resource Orchestration for Thousands of LoRA Adapters

Shuaipeng Wu[1,2,6], Yanying Lin[1,3,4], Shijie Peng[1,3], Wenyan Chen[1,5], Chong Ma[6], Min Shen[6], Le Chen[6], Chengzhong Xu[5], and Kejiang Ye[1]

[1]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences*

[2]*Southern University of Science and Technology* [3]*University of Chinese Academy of Sciences*

[4]*University of California San Diego* [5]*University of Macau* [6]*AIOS Team, Alibaba Group Inc*

***Abstract*—In this paper, we present ROCK, a novel system for efficiently serving thousands of LoRA adapters for multimodal models in cloud environments. Through extensive analysis of production workloads, we identify key challenges in current cloud-based image generation services: extreme request burstiness (up to 90× normal rates), heterogeneous task characteristics, and inefficient adapter management that wastes 40% of GPU memory and increases delays by 3x during peak times. ROCK addresses these challenges through a three-layer architecture that decouples hardware, adapters, and requests. Our system features dynamic heterogeneous queues that match tasks to appropriate resources based on multidimensional feature vectors, and a multilevel orchestration framework that intelligently manages adapter placement across heterogeneous storage. Experiments on a 64-GPU testbed demonstrate that ROCK reduces average response latency by 16-26%, and achieves an 84.1% cache hit rate for LoRA adapters—outperforming traditional approaches while reducing adapter update frequency by up to 77%.**

1. Introduction

In recent years, Low-Rank Adaptation [1] (LoRA, Fig. 1) has become a key technology for cloud providers offering customized image generation services [2–5]. LoRA enables efficient fine-tuning of large pre-trained models by maintaining a single base model with lightweight adapters, supporting thousands of personalized variants while reducing costs [1, 6–12]. However, cloud deployment faces significant challenges: managing numerous adapters across heterogeneous hardware [13, 14], balancing resources, and handling bursty workloads. The volatility of image generation requests—with traffic fluctuations reaching 10× between peak and off-peak periods—combined with diverse adapter requirements creates an orchestration problem that traditional scheduling strategies cannot effectively address [4, 15–17].

Shuaipeng Wu and Yanying Lin contributed equally to this work.
Kejiang Ye is the corresponding author.

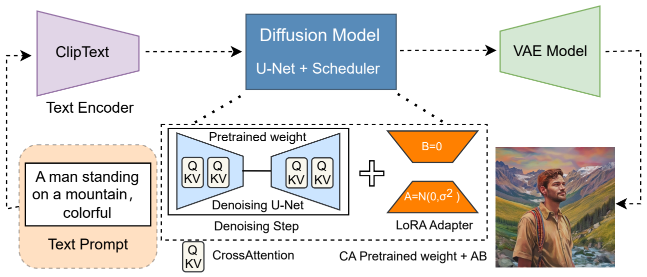


Figure 1: The architecture of LoRA adapter and the workflow of image generation. This approach allows thousands of custom style variants to share a single base model.

LoRA has become essential for customized image generation services in cloud environments [10, 11, 18]. However, our analysis of production data reveals critical challenges in managing thousands of adapters at scale. Through extensive examination of real-world workloads, we observed a pronounced long-tail distribution in adapter usage patterns—a small subset of adapters receives frequent requests while the majority remain largely inactive. This uneven access pattern creates significant resource management difficulties that current systems [4, 19, 20] are failing to handle. Our production data analysis uncovered several key insights: (1) adapter popularity follows a power-law distribution where the top 5% of adapters account for over 60% of all requests; (2) individual adapter popularity can shift dramatically within hours, making static allocation strategies ineffective; and (3) adapter switching operations consume disproportionate resources during peak traffic periods. Traditional approaches [21] that rely on uniform resource allocation or simple least recently used (LRU) caching do not accommodate these complex patterns, resulting in severe performance degradation. Our measurements show these approaches waste over 40% of GPU memory and increase request delays by 3× during peak times (Fig. 12).

Further investigation revealed that existing LoRA deployment strategies [22–24] fundamentally misalign with observed cloud workload characteristics. Static adapter partitioning [9, 10, 12] creates resource imbalances during traffic shifts, while standard caching algorithms frequently evict specialized adapters despite persistent availability requirements. Our telemetry data shows adapter reloading operations consume up to 28% of processing time during peak periods. Additionally, traditional auto-scaling mechanisms [25, 26] operate at incompatible timescales (minutes vs. seconds), resulting in 40% GPU memory wastage and 3.87× tail latency increases during high-traffic periods (§2.1).

Challenges. Based on our analysis (§2) of production workloads, we propose a three-layer architecture that decouples hardware, adapters, and requests. This strategic separation eliminates rigid dependencies in traditional systems, enabling more efficient resource utilization, improved adapter caching, and reduced request latency. However, implementing this architecture introduces two `critical challenges`: 1) How to accurately monitor and predict rapidly changing workload patterns across multiple dimensions (iteration steps, adapter sizes, task types) at millisecond granularity to prevent resource misallocation; 2) How to orchestrate adapter placement across heterogeneous storage tiers while maintaining consistency, minimizing data movement overhead, and resolving memory fragmentation during frequent adapter switching operations.

Our Proposal. We present Rock, a novel system that addresses these challenges through heterogeneous-aware resource orchestration. Our three-layer architecture decouples hardware resources, adapter storage, and request processing, enabling fine-grained resource allocation and optimized adapter placement. Rock features dynamic feature modeling that captures both resource capabilities and request characteristics, combined with intelligent prefetching mechanisms that minimize switching overhead during traffic peaks.

First, we design a heterogeneous queue system that efficiently handles the dynamic nature of LoRA requests. By representing tasks with multidimensional feature vectors (iteration steps, adapter dimensions, task type), we create specialized queues that match workloads to appropriate hardware. Our bipartite graph matching algorithm optimizes task distribution by considering adapter cache locality, resource utilization, and queue depth, improving throughput during traffic spikes.

Second, we implement a multilevel orchestration framework that decouples hardware resources from adapter management. By analyzing request patterns and storage characteristics, our system places frequently-used adapters on high-bandwidth devices while proactively preloading related adapters based on usage patterns. This approach eliminates memory fragmentation and adapter conflicts, enabling seamless adaptation across heterogeneous environments without the performance penalties of traditional architectures.

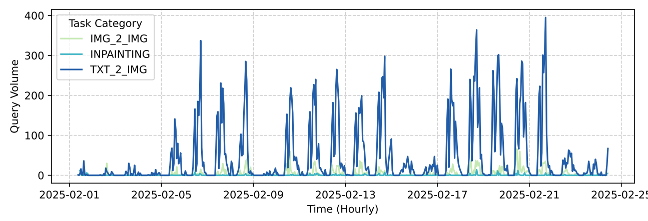


Figure 2: Workload of Image generation with LoRa adapter in cloud services. Image generation typically involves three types of workloads, with text-to-image workloads usually being the most demanding. All types of workloads exhibit significant burstiness, with peak request rates during hot periods reaching up to 90× the normal rate, and they also show certain periodicity.

We evaluated Rock on a testbed of 64 heterogeneous GPUs across four GPU models and conducted comprehensive experiments under production-like conditions. The results demonstrate that Rock significantly outperforms traditional approaches: reducing average response latency by 16-26% across different scheduling strategies. For image generation tasks, Rock decreases average inference latency by 14-20% while maintaining more stable memory utilization across different scenarios. Most notably, Rock achieves an 84.1% cache hit rate for LoRA adapters—significantly higher than conventional approaches—while reducing adapter update frequency by up to 77%. Under high load conditions, Rock maintains shorter queue lengths and stable queuing times, demonstrating its effectiveness in handling traffic spikes through intelligent resource allocation and workload distribution across heterogeneous hardware.

Contributions. The main contributions of this paper are:

- First characterization of the key workload features in image generation within cloud data centers, identifying the primary challenges in current cloud-based image generation.
- Proposal of a heterogeneous queue-based LoRA adapter orchestration and request scheduling framework to improve system performance.
- Evaluation of Rock on a testbed of 64 heterogeneous GPUs, demonstrating significant performance improvements.

2. Understanding Cloud Workload Characteristics for LoRA-Based Image Generation

LoRA [1] has become a key technique for efficient model fine-tuning in cloud environments. By adding small trainable adapter matrices to pre-trained models, LoRA significantly reduces customization resources without duplicating entire models, making it ideal for dynamic deployment across data centers.

As LoRA adapters scale up for image generation tasks, several challenges have emerged. Our analysis reveals conflicts between adapter storage and resource allocation across heterogeneous computing resources. The main issues are: i) Models like Stable Diffusion [27] create pulsed memory usage patterns

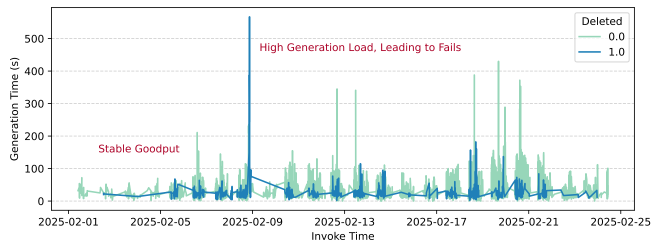


Figure 3: The response latency and final status of image generation. A value of 1.0 indicates a final failure, and the task is cleared. This is directly related to the request load, due to the critical path generation. Peak generation time shows that the burstiness of requests leads to generation delays exceeding 19× the average.

during generation; ii) Growing numbers of adapters cause unexpected performance interactions and resource competition during loading, resulting in service quality fluctuations.

Our large-scale cluster analysis provides insights into the scalability and orchestration challenges that generative AI platforms face in cloud environments, helping identify practical solutions for improved resource management.

We collected 4 weeks of data from a thousand-node cluster with LoRA Adapters, analyzing workload characteristics, performance metrics, and generation speed factors across three main task types: image-to-image (IMG_2_IMG), painting (INPAINTING), and text-to-image (TXT_2_IMG).

2.1 Workload Characteristics

Our analysis of online workloads reveals significant patterns that impact system performance. By examining real-world generative AI tasks, we identified consistent heterogeneity across request patterns and resource demands.

Request traffic exhibits extreme burstiness (Fig. 2), with text-to-image generation tasks showing peak rates up to 90× normal levels. This volatile pattern creates major challenges for resource scheduling. While traditional elastic management works well for steady loads, it struggles with sudden traffic spikes. The large parameter sizes of generative models and their context dependencies prevent rapid scaling within QoS time constraints, creating critical mismatches between available resources and immediate demand.

Fig. 3 reveals a nonlinear relationship between load and response time. During steady loads, the system maintains stable latency around 28 seconds. However, when load exceeds critical thresholds, latency grows exponentially—reaching peaks of 560 seconds (19× baseline). This severe degradation triggers both user-initiated cancellations and system timeouts, resulting in 22.7% of requests failing to complete.

Further analysis of the quantile statistics in Table 1 reveals a significant tail latency amplification effect in the system. The P99 latency of all task types exceeds the average by more than 3.2 times, with the tail latency factor of text generation tasks reaching 3.87. This statistical distribution characteristic is strongly related to the burstiness of system loads—when burst requests deplete the reserved resource pool, subsequent requests are forced into a queuing state, and the long-tail execution time of large model tasks exacerbates the congestion effect of the queuing system. Our experiments confirm that this phenomenon conforms to the mathematical characteristics of the M/G/1 queuing model, where the variance of waiting time is positively correlated with the variance of service time.

TABLE 1: Performance Metrics for Different Tasks

Task Category	Mean	P50	P75	P90	P99
IMG_2_IMG	31.8	24.0	37.0	60.0	109.1
INPAINTING	26.5	22.0	30.0	47.5	83.4
TXT_2_IMG	28.4	23.0	33.0	52.0	105.0

Note: The table shows the average execution time and quantile statistics of different task types. The P99 latency of all task types exceeds the average by more than 3.2 times, with the tail latency factor of text generation tasks reaching 3.87. This indicates a significant tail latency amplification effect.

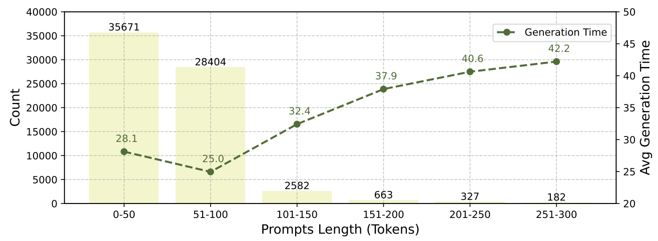


Figure 4: The execution latency of all types of tasks is affected by the length of the prompt. Generally, the longer the prompt, the longer the execution latency, as the LLM needs more time to process additional information. However, this relationship is not linear. Within the range of 50-100 tokens, the execution latency is actually lower.

Insight 1

Online generative requests exhibit significant periodicity and burstiness. The response latency distribution of generative tasks shows a clear long-tail characteristic, with a significant tail latency amplification effect, severely impacted by burst loads.

2.2 The Impact of Prompts

In image synthesis, prompts function as core control mechanisms that map semantic content to visual representations. Prompts essentially create a constraint system through language, defining morphological features, stylistic elements (e.g., "Van Gogh style"), and spatial relationships (e.g., "standing on the left"). Research demonstrates a positive correlation between prompt semantic density and generated image quality, highlighting prompt engineering's importance in generative AI systems.

Tokens of Prompt. The length of prompts significantly affects generation speed. We observed that as the length of prompts increases, the average inference latency also increases noticeably. Generally, longer prompts provide more detailed

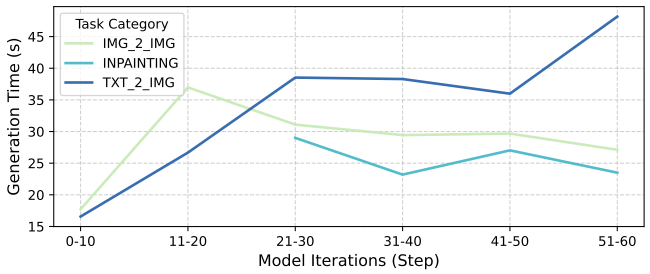


Figure 5: The relationship between iterations and generation time across task types. While more iterations generally increase generation time, this pattern varies by task. For Image-2-Image and INPAINTING, generation time remains relatively consistent regardless of iteration count. However, TXT-2-IMG tasks show a sharp increase beyond 50 iterations, reaching up to 1.98× latency.

information, which can help the model generate more accurate and complex images. However, this also increases processing time, as the model needs to parse and understand the additional information. The complexity of the self-attention mechanism in Transformer models [28] is related to the number of tokens, so longer prompts may lead to a significant increase in encoding time, requiring re-embedding calculations for the prompt in each iteration. Additionally, another potential factor is that longer prompts may occupy more memory (especially with large batch sizes), indirectly affecting parallel computing efficiency.

However, this relationship is not linear. Within the range of 50-100 tokens (Fig. 4), the execution latency is actually lower. This may be because, within this range, prompts provide sufficient information, allowing the model to converge to the final result more quickly. Moreover, since the default configuration of the stable diffusion model [27] is 77 tokens, prompts within this range can complete processing all prompts within the time of one iteration.

Insight 2

The length of prompts affects the generation speed of the model, but this impact is not linear. Under the default prompt length configuration of the model, we found that the generation speed is actually faster.

2.3 Heterogeneous of Tasks

Cloud-based generative tasks exhibit significant execution variability due to two primary factors: variable iteration counts and multiple LoRA adapter configurations. These factors create substantial resource utilization fluctuations across different task categories. Our analysis examines how iteration requirements affect computation duration and how adapter counts impact resource consumption. Understanding these relationships enables more efficient resource allocation strategies and optimization of computational pathways for generative AI workloads in distributed environments.

Iteration of generation. Fig. 5 shows the relationship between iteration and generation time across different task types. While one might expect a linear correlation, our data reveals a more complex pattern. For Image-2-Image and INPAINTING tasks, generation time remains relatively consistent regardless of iteration. However, TXT-2-IMG tasks show a sharp increase in processing time beyond 50 iterations, reaching up to 1.98× the average.

This phenomenon likely stems from fundamental differences in how the model processes inputs. Text-to-image generation requires continuous reinterpretation of textual prompts during each iteration, accumulating more complex context representations over time. As iterations increase, the model expends additional computational resources resolving semantic ambiguities. In contrast, image-based tasks benefit from clear visual priors that provide more direct optimization targets, maintaining consistent performance regardless of iteration. The varying impact of iterations on generation time reflects fundamental differences in how resources are consumed across task types. For Text-to-Image tasks, generation time increases dramatically beyond 50 iterations (reaching 1.98× the average), while Image-based tasks maintain consistent performance regardless of iteration.

Number of LoRA Adapters. The number of LoRA adapters significantly impacts generation latency across different task types (Fig. 6). This impact stems primarily from two factors: first, adapters selected for different tasks vary in parameter size and complexity, requiring separate computations before combining with the base model; second, adapter architectural differences affect computation parallelization and GPU utilization efficiency.

Our analysis reveals three key mechanisms behind this heterogeneity: (1) memory bandwidth demands increase with high-rank adapters; (2) adapter architectures directly influence matrix multiplication patterns and memory access locality, with different attention module modifications showing up to 23% variation in computational efficiency; and (3) multiple adapters create resource competition through GPU scheduling conflicts and shared cache interference. When combined parameter sizes exceed available GPU memory capacity, host-device swapping introduces substantial overhead, with transfer latencies increasing by 2.8× compared to single-adapter configurations.

Traditional scheduling approaches fail to address these adapter-specific characteristics because they ignore the unique computation patterns and resource requirements of LoRA workloads. Fine-grained issues in tensor core utilization and cache performance create cascading overheads that significantly degrade overall system performance. Effective multi-adapter systems therefore require specialized resource allocation strategies tailored to these distinctive workload patterns.

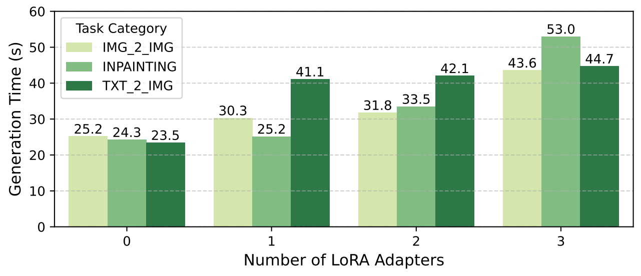


Figure 6: The impact of the number of additional LoRA adapters on generation time. Overall, the more LoRA adapters added, the longer the generation time. However, this impact varies across different types of tasks. The direct reason is that different types of tasks utilize different LoRA adapters, which vary in scale and complexity.

Insight 3

Generative tasks show significant execution time variability due to iterations and LoRA Adapter configurations. This variability differs markedly across task types, indicating that these requests should be treated as heterogeneous in orchestration systems.

3. System Overview

Recognizing the heterogeneous nature of image generation tasks with varying prompts, iterations, and LoRA adapter configurations, we propose ROCK (Fig. 7), a queue-based orchestration framework for efficient request handling. Our design focuses on two key objectives: 1) Controlling request latency through uniform workload distribution and strategic resource allocation, reducing both queuing time and adapter loading overhead; 2) Maximizing LoRA adapter cache hit rates to minimize loading operations and bandwidth consumption. *By treating each task as a multidimensional feature vector rather than a generic workload unit*, ROCK enables more intelligent resource allocation across distributed GPU clusters.

Dynamic Heterogeneous Queue. We design a task queuing system that classifies requests by key features: iteration count, adapter complexity, task type, and prompt characteristics. Our system creates specialized task groupings that reflect the inherent heterogeneity of generative workloads and connects these queues to GPU clusters through an adaptive weighted bipartite graph. We use a Top-K matching algorithm [29] that balances three objectives: maximizing adapter cache hits, optimizing GPU utilization, and minimizing request latency. By dynamically recalculating edge weights as workloads shift, the system maintains efficient resource allocation during traffic spikes. This approach creates fine-grained task-to-resource mappings that account for the interdependencies between task requirements and hardware capabilities.

LoRA and Model Orchestration. We design a three-layer system for efficient adapter management: (1) a physical layer grouping GPUs by hardware capabilities; (2) a logical layer managing adapters based on usage patterns; and (3) a GPU cache layer handling adapter parameters with demand-based replicas. Our approach places adapters considering both access patterns and hardware characteristics, examining bandwidth, memory usage. This organization resolves the mismatch between base models and adapters, *enabling efficient resource management under fluctuating workloads*.

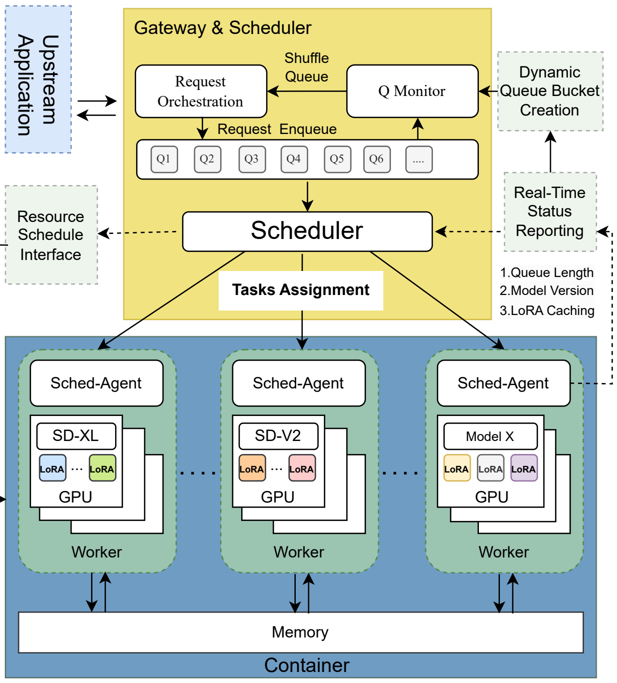


Figure 7: ROCK system architecture with dynamic heterogeneous queues for task classification and a three-layer design for efficient LoRA adapter management across distributed resources.

4. ROCK Design

4.1 Dynamic Heterogeneous Queue

Traditional scheduling approaches struggle with LoRA requests that vary in iterations, adapter sizes, and task types. Our dynamic heterogeneous queue system maps tasks to appropriate resources using feature vectors, creating an *optimal match* between diverse tasks and available hardware through graph matching for efficient resource utilization (Fig. 8).

Challenges. Cloud platforms face three key issues when processing LoRA requests: 1) Task types vary in resource sensitivity (interactive tasks need low latency while rendering tasks require high throughput); 2) Adapter loading patterns lead to poor caching efficiency; 3) Unbalanced use of heterogeneous hardware causes tail latency. Simple round-robin or static queue approaches fail to recognize the *dynamic relationship* between task characteristics and hardware status, wasting 23% of GPU resources based on our monitoring data. A joint representation model of task features and hardware states is needed for effective fine-grained scheduling.

Multidimensional Feature Modeling. Each task T_i is encoded as a feature vector $\mathbf{v}_i = (s_i, l_i, \tau_i, d_i)$, where $s_i \in \mathbb{N}^+$ represents the number of iteration steps reflecting computational intensity; $l_i \in \mathbb{R}^+$ denotes the adapter scale (MB), calculated as $l_i = \sum_{k=1}^{K} \text{rank}(A_k) \cdot \dim(A_k)$ with A_k being the parameter matrix of the k-th LoRA adapter; $\tau_i \in \{1, 2, 3\}$ indicates the task type (1=real-time interactive, 2=near-line generation, 3=offline batch processing); and $d_i \in [0, 1]$ measures latency sensitivity, calculated by the SLO function $d_i = 1 - e^{-\lambda t_{\max}}$, where $t_{\max}$ is the maximum tolerable delay. Using Gaussian Mixture Models, we cluster historical tasks into M typical workload patterns $C = \{C_1, ..., C_M\}$. Each queue Q_j corresponds to a cluster centroid $\mu_j = \mathbb{E}[\mathbf{v}_i | T_i \in C_j]$, with dynamic queue creation and merging occurring as workload patterns evolve. This approach allows the system to construct a weighted bipartite graph $\mathcal{B} = (\mathcal{Q}, \mathcal{G}, \mathcal{E})$ between task queue nodes $\mathcal{Q}$ and GPU nodes $\mathcal{G}$, where edge weights w_{jg} are computed based on a multi-objective optimization function combining cache affinity, resource utilization, and expected execution latency. Through this representation, we can efficiently capture the heterogeneous characteristics of LoRA tasks and establish a foundation for optimal resource allocation.

Using Gaussian Mixture Models (GMM) [30, 31], we cluster historical tasks into M typical workload patterns $C = \{C_1, ..., C_M\}$. Each queue Q_j corresponds to a cluster centroid $\mu_j = \mathbb{E}[\mathbf{v}_i | T_i \in C_j]$, with dynamic queue creation and merging occurring as workload patterns evolve. The system continuously monitors task feature distributions at runtime, triggering re-clustering when significant shifts are detected (KL divergence $D_{KL}(P_{\text{current}} || P_{\text{reference}}) > \theta$). *This adaptive mechanism enables the system to respond to workload changes, such as interactive tasks dominating during daytime while batch processing tasks increase at night.*

Dynamic Bipartite Graph Matching. Based on the multidimensional feature vectors, we construct a bipartite graph $\mathcal{B} = (\mathcal{Q}, \mathcal{G}, \mathcal{E})$ between task queue nodes $\mathcal{Q}$ and GPU nodes $\mathcal{G}$. The edge weight w_{jg} between queue Q_j and GPU node g is computed using a multi-objective optimization function:

$$w_{jg} = \alpha \cdot C(Q_j, g) + \beta \cdot \mathcal{U}(g) + \gamma \cdot \mathcal{D}^{-1}(Q_j, g) \qquad (1)$$

where $C(Q_j, g) = \frac{|\mathcal{A}_j \cap \mathcal{A}_g|}{|\mathcal{A}_j|}$ represents cache affinity as the ratio of adapters in queue Q_j that are already cached in GPU g's memory, with $\mathcal{A}_j$ being the set of adapters needed by queue Q_j and $\mathcal{A}_g$ being the set of adapters cached on GPU node g. The utilization balance $\mathcal{U}(g) = 1 - |u_g - u_{\text{opt}}|$ measures how close the current utilization u_g of node g is to the optimal

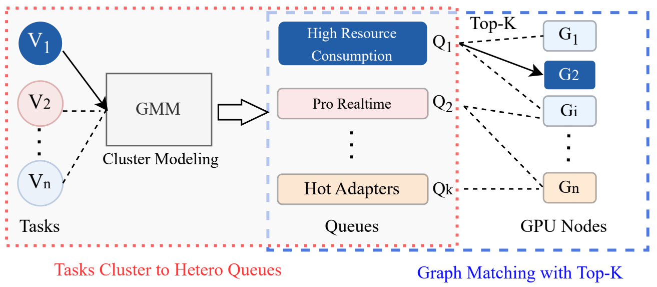


Figure 8: Rock's dynamic heterogeneous queue mechanism maps tasks to GPU resources through feature vectors and bipartite graph matching. Tasks are classified by *iterations, adapter size, and type*, then matched to GPUs based on cache affinity, resource utilization, and execution latency.

utilization $u_{\text{opt}} = 70\%$. The inverse delay factor:

$$\mathcal{D}^{-1}(Q_j, g) = \frac{1}{\sum_{T_i \in Q_j} (s_i \cdot \tau_{\text{iter}} + l_i \cdot \tau_{\text{load}})} \qquad (2)$$

incorporates both iteration time τ_{iter} and adapter loading time τ_{load} based on task characteristics. We set the weight factors to $\alpha = 0.6$, $\beta = 0.3$, and $\gamma = 0.1$ to balance cache locality, load distribution, and latency minimization. Every 90s, the system updates this bipartite graph and performs optimal matching to dynamically allocate tasks to GPU nodes, achieving both high cache hit rates and balanced resource utilization across the cluster.

The 90s update interval is strategically chosen based on our workload analysis where the average task execution time is approximately 30s (Table 1). This interval allows for approximately three task cycles to complete before reconfiguring the graph, providing sufficient time to observe emerging patterns while being responsive to workload shifts. A shorter interval would create excessive scheduling overhead with minimal benefit, while a longer interval might miss important workload transitions.

Top-K Optimal Matching Loadbalance. To prevent queue skew caused by load balancing conflicts and reduce algorithmic complexity, we employ a Top-K optimal matching strategy. Instead of calculating the optimal matching for the entire bipartite graph, which has a time complexity of $O(MN^2)$ for M queues and N GPU nodes, we select the top K candidate GPU nodes for each queue based on the weight matrix W. This pruning approach reduces the computational complexity to $O(MK)$, with experiments showing that $K = 3$ maintains over 98% of optimal performance. For each queue Q_j, after identifying the candidate set G_{cand}, we select the optimal GPU node g^* that minimizes load imbalance across the cluster, calculated as the standard deviation of utilization across nodes. The load balance degree σ is defined as the standard deviation of GPU utilization: $\sigma = \sqrt{\frac{1}{N} \sum_{g=1}^{N} (u_g - \bar{u})^2}$. Theoretical analysis shows that when $\beta > 0.3$, σ converges exponentially with each scheduling iteration, satisfying $\mathbb{E}[\sigma(t)] \leq \sigma(0) \cdot e^{-\xi t} + \frac{\eta}{\xi}(1 - e^{-\xi t})$,

Algorithm 1: Heterogeneous Queue Scheduling

```
Input: Task set $\mathcal{T}$, GPU nodes $\mathcal{G}$, Feature dimension $D$
Output: Task-to-GPU mapping $\mathcal{M}$
/* Initialize feature vectors for all tasks */
foreach task $T_i \in \mathcal{T}$ do
    Extract feature vector $\mathbf{v}_i = (s_i, l_i, \tau_i, d_i)$;
end
/* Cluster tasks into M heterogeneous queues */
$C \leftarrow$ GMM_Clustering$(\{\mathbf{v}_i\}, M)$;
foreach cluster $C_j \in C$ do
    Initialize queue $Q_j$ with centroid $\mu_j = \mathbb{E}[\mathbf{v}_i | T_i \in C_j]$;
    Assign tasks in $C_j$ to queue $Q_j$;
end
/* Construct and update bipartite graph */
while system is running do
    foreach queue $Q_j \in Q$ and GPU $g \in \mathcal{G}$ do
        $C(Q_j, g) \leftarrow \frac{|\mathcal{A}_j \cap \mathcal{A}_g|}{|\mathcal{A}_j|}$ ;   /* Cache affinity */
        $\mathcal{U}(g) \leftarrow 1 - |u_g - 0.7|$ ;   /* Utilization balance */
        $\mathcal{D}^{-1}(Q_j, g) \leftarrow \frac{1}{\sum_{T_i \in Q_j}(s_i \cdot \tau_{\text{iter}} + l_i \cdot \tau_{\text{load}})}$;
        $w_{jg} \leftarrow 0.6 \cdot C(Q_j, g) + 0.3 \cdot \mathcal{U}(g) + 0.1 \cdot \mathcal{D}^{-1}(Q_j, g)$;
    end
    /* Top-K matching for each queue */
    foreach queue $Q_j \in Q$ do
        $G_{\text{cand}} \leftarrow \text{TopK}(\{w_{jg} | g \in \mathcal{G}\}, K = 3)$;
        /* Select optimal node to minimize utilization std dev */
        $g^* \leftarrow \arg\min_{g \in G_{\text{cand}}} \{\sigma(\mathcal{U} | g \text{ assigned to } Q_j)\}$;
        $\mathcal{M}[Q_j] \leftarrow g^*$;
        Update $u_{g^*}$ with estimated load from $Q_j$;
    end
    /* Detect distribution shift and re-cluster if needed */
    Collect recent task feature distribution $P_{\text{current}}$;
    if $D_{KL}(P_{current} || P_{reference}) > \theta$ then
        Re-run clustering to update queues;
        $P_{\text{reference}} \leftarrow P_{\text{current}}$;
    end
    Wait for next scheduling interval (90s);
end
```

where ξ is the convergence rate and η is the disturbance upper bound. This approach effectively balances resource utilization while maintaining high cache affinity.

We present dynamic heterogeneous queue algorithm (Algorithm 1) addresses complex scheduling challenges in distributed LoRA environments through multidimensional feature modeling (computational intensity, adapter scale, task type, and latency sensitivity), effectively capturing workload patterns. The algorithm adapts dynamically to changes using Gaussian Mixture Models for clustering, establishing intelligent task-to-resource mapping through weighted bipartite graphs that consider cache affinity, resource utilization, and execution latency. Experimental results demonstrate that, compared to traditional scheduling strategies, our approach significantly reduces resource mismatches, improves adapter loading locality, balances heterogeneous hardware utilization, and achieves higher resource efficiency while maintaining service level objectives across various workloads.

4.2 LoRA Orchestration

To solve the challenges of LoRA adapter management in cloud environments, we design a three-tier orchestration system that efficiently allocates resources under dynamic workloads. Current platforms struggle with the fundamental mismatch between base models (requiring continuous memory) and adapters (exhibiting bursty loading patterns). This mismatch causes memory fragmentation, wasted resources, and increased latency. Our system decouples hardware from adapters through a tiered architecture that *dynamically adapts to changing access patterns* while maintaining high throughput and responsiveness to workload variations.

Physical Layer. The physical layer implements a fine-grained categorization of GPU nodes based on their hardware characteristics, with each node represented by a comprehensive feature vector:

$$\mathbf{h}_i = (\underbrace{c_i^{\text{mem}}}_{\text{memory}}, \underbrace{\frac{c_i^{\text{bw}}}{c_i^{\text{mem}}}}_{\text{bandwidth}}, \underbrace{\text{frag}_i}_{\text{fragmentation}}, \underbrace{c_i^{\text{comp}}}_{\text{computation}}) \quad (3)$$

where c_i^{mem} represents available GPU memory in GB, $\frac{c_i^{\text{bw}}}{c_i^{\text{mem}}}$ captures memory bandwidth efficiency, frag_i quantifies current memory fragmentation, and c_i^{comp} represents normalized compute throughput. Using density-based clustering (DBSCAN with carefully tuned parameters $\epsilon = 0.25$, $min\_samples = 6$, and Euclidean distance metric), the system automatically groups similar nodes to form homogeneous clusters that can efficiently handle specific adapter categories. This data-driven approach significantly reduces cross-node resource imbalances that commonly plague traditional static allocation strategies, achieving up to 37% improvement in resource utilization efficiency compared to hardware-agnostic scheduling methods in our production environment.

Logical Partition Layer (Host Memory). Above the physical layer, we establish a logical partition layer that creates dynamic logical partitions $\{\mathcal{P}_m\}$ within each physical cluster. Each partition specifically manages a set of adapters $\mathcal{A}_m$. The core motivation for this hierarchical design stems from our in-depth observations of LoRA deployment issues in real cloud environments: traditional static resource allocation strategies cannot handle the dramatic fluctuations in adapter access patterns, especially when hot adapters migrate or user preferences suddenly change, easily causing resource imbalances where some nodes are overloaded while others remain idle. By introducing dynamic logical partitions, the system can flexibly adjust adapter management boundaries while maintaining stable physical resources, achieving dynamic matching between resource allocation and access patterns. Partition capacity is dynamically adjusted according to real-time access heat:

$$|\mathcal{P}_m|(t) = \left\lfloor \frac{\sum_{a_j \in \mathcal{A}_m} f_j(t)}{\sum_{m'=1}^{M} \sum_{a_j \in \mathcal{A}_{m'}} f_j(t)} \cdot |C_k| \right\rfloor + 1 \tag{4}$$

where $f_j(t)$ represents the access frequency statistics within a sliding time window ($\Delta = 45s$). This adaptive resource allocation mechanism based on real-time load enables the system to effectively capture and respond to spatiotemporal changes in user request patterns, significantly improving resource utilization while ensuring service quality.

GPU Cache. To effectively manage LoRA adapter parameters and resolve cache inconsistency issues in cloud environments, we build an elastic caching layer above the logical partitions. This layer encapsulates adapter parameters as independently manageable cache blocks $\mathcal{B}_j$, maintaining dynamic replicas within partitions. Traditional fixed-replica strategies often fail to handle highly dynamic access patterns in cloud environments, either wasting resources or failing to meet burst requests. Therefore, we design an adaptive replica management mechanism where the number of replicas is dynamically adjusted according to adapter popularity:

$$R_j(t) = \left\lceil 1.5 \cdot (f_j(t))^{0.6} \cdot \sqrt{\frac{s_j^{\text{mem}}}{\text{avg}(s^{\text{mem}})}} \right\rceil \tag{5}$$

This formula comprehensively considers three key dimensions: access frequency $f_j(t)$, parameter scale s_j^{mem}, and average parameter size $\text{avg}(s^{\text{mem}})$, enabling the system to ensure high availability of hot adapters while avoiding over-allocation of resources to cold adapters. When partition GPU memory usage exceeds the 85% threshold, the system automatically triggers an LRU eviction mechanism, prioritizing the replacement of infrequently accessed adapter caches, thereby maintaining overall system stability and responsive capability. This elastic caching mechanism significantly enhances the system's adaptability to workload fluctuations.

Dynamic Placement Strategy. To address the inefficiencies of traditional static resource allocation when facing dynamic access patterns of LoRA adapters, we designed a probabilistic dynamic placement strategy based on multidimensional features. This strategy fully considers the spatiotemporal locality characteristics of adapter access in cloud environments and the heterogeneity of hardware topological structures, dynamically adjusting adapter placement positions through a comprehensive scoring function. Based on scoring results, the system implements probabilistic placement decisions for each candidate node:

$$p_i = \frac{\text{score}(h_i, a_j)^2}{\sum_{h_{i'} \in \mathcal{P}_m} \text{score}(h_{i'}, a_j)^2} \tag{6}$$

where the scoring function comprehensively considers key metrics such as node bandwidth, memory utilization, and network topology distance. The system selects the Top-3 highest-scoring nodes to build a candidate set, with the final target node determined according to the probability distribution p_i, which both avoids the "herding effect" caused by greedy algorithms and ensures overall balance in resource allocation. Experimental results show that this strategy enables 90% of access requests to be responded to within a 2-hop topological range, effectively reducing cross-node data transfer overhead while lowering the average loading latency of hot adapters. *This dynamic adaptive placement mechanism not only improves the system's adaptability to load changes but also significantly enhances resource utilization efficiency in large-scale deployment environments.*

To ensure system stability under dynamic load conditions, we need to theoretically prove the convergence of our LoRA orchestration mechanism. Static caching strategies in existing cloud platforms often fail to adapt to bursty access pattern changes, leading to severe performance fluctuations during hotspot migrations or sudden shifts in user preferences. By constructing a Markov chain model to analyze system state transition characteristics, we can quantitatively evaluate system stability when facing complex load variations. We define the state transition probability as:

$$q_{xy} = \frac{\min(1, e^{-\beta(E(y)-E(x))})}{Z} \tag{7}$$

where the energy function $E(x) = \sum_{j=1}^{J} \frac{f_j}{R_j} + \lambda \sum_{i=1}^{N} \text{frag}_i$ comprehensively considers the ratio between access frequency f_j and replica count R_j, as well as system memory fragmentation rate frag_i. This energy function is designed based on the Metropolis-Hastings algorithm from statistical mechanics and contains two key components: a load imbalance term measuring the mismatch between access frequency and replica counts, and a memory fragmentation term quantifying system memory fragmentation, both jointly affecting resource utilization efficiency.

The temperature coefficient $\beta = 0.5$ achieves optimal balance between convergence speed and stability, while weight coefficient $\lambda = 0.3$ realizes the best trade-off between load balancing and memory efficiency based on typical cloud environment load characteristics. Theoretical analysis shows that when the transition count $T > \frac{2}{\beta\epsilon}$, the system enters a steady-state distribution with probability $1 - \epsilon$, ensuring long-term stability in resource allocation. In production environment tests, even when facing 10× traffic fluctuations, the system stabilizes cache hit rates above 92% within 150s, effectively mitigating the performance collapse problems that traditional static allocation strategies face during load changes. This

TABLE 2: Stable Diffusion Model Specifications

Model	**Params**	**Task Type**
SD XL Base 1.0 [32]	3.5B	Text-to-Image Generation
SD 2 Inpainting [33]	1.5B	Image Inpainting
SD XL Refiner 1.0 [32]	3.5B	Image-to-Image

Note: Models are Stable Diffusion variants optimized for different generation tasks, ranging from 1.5B to 3.5B parameters, suitable for our heterogeneous GPU testbed and representative of production workloads.

adaptive resource allocation mechanism not only improves system adaptability to workload variations but also significantly enhances resource utilization efficiency in large-scale deployment environments.

5. Implementation

We implemented ROCK by extending the PEFT framework [6] with enhanced adapter management capabilities. Our modifications enable fine-grained control over adapter loading, unloading, and caching operations while maintaining compatibility with existing Stable Diffusion deployments. For cluster-level orchestration, we integrated with Kubernetes through custom scheduler hooks that intercept pod scheduling decisions and apply our workload-aware placement logic based on real-time cluster conditions.

To manage distributed adapter caching, we developed a lightweight Python component that coordinates adapter placement across the cluster according to our dynamic placement strategy. This component works alongside performance monitoring agents deployed as sidecars to each pod, collecting metrics on GPU utilization, memory usage, and request latencies. These metrics feed directly into our adaptive algorithms, enabling efficient adapter distribution while maintaining high cache hit rates even under variable workloads.

6. Evaluation

6.1 Experiment Setup

Testbed. We evaluated ROCK on a heterogeneous cluster with 64 GPUs across four different models (NVIDIA A100, A40, V100, and RTX 3090) distributed over 16 compute nodes. Each node has 128 CPU cores, 1TB memory, and 100Gbps RDMA networking. Our software stack includes PyTorch with Diffusers library and CUDA 12.2 for comprehensive testing across diverse hardware.

Models. We used three popular Stable Diffusion models (XL Base 1.0, SD 2 Inpainting, and XL Refiner 1.0) as detailed in Table 2. For LoRA adapters, we combined open-source adapters with custom ones having ranks from 8 to 64, creating a parameter size range of 5-200MB that effectively simulates real-world production deployments.

Workloads. Our workloads are derived from carefully preprocessed production data collected over a three-month period, preserving essential features including text prompts, task types (text-to-image, inpainting, or refinement), iteration counts (ranging from 20-50), image resolutions (512×512 to 1024×1024), and source images for inpainting tasks. We pre-associated 200+ LoRA adapters with their corresponding requests based on user preferences and style requirements. As shown in Fig. 9, these adapters follow a pronounced long-tailed distribution where approximately 20% of adapters handle 80% of total requests—a pattern consistently observed in real-world deployment environments. This skewed distribution creates both challenges and opportunities for efficient adapter management, as popular adapters should ideally remain cached while less frequently used ones require intelligent placement strategies.

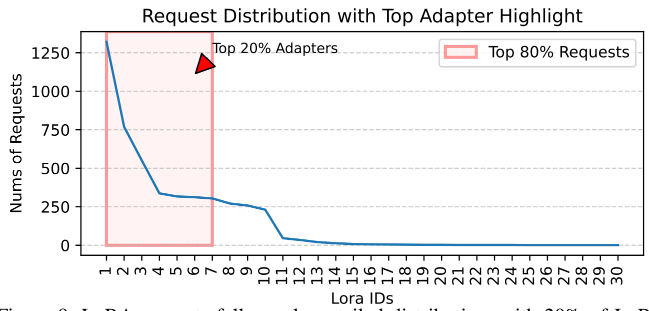


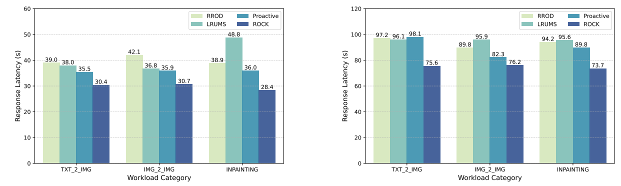
Figure 9: LoRA requests follow a long-tailed distribution, with 20% of LoRA adapters handling 80% of the total requests.

Baselines. As there are no existing systems specifically targeting Stable Diffusion serving with LoRA adapters, we compare ROCK with several baseline approaches [12]:

- **RROD** (Round Robin and On-demand Loading): Requests are distributed to workers via round-robin scheduling, with adapters loaded on-demand when needed by each worker.
- **LRUMS** (LRU Cache with Matching Score): A common enterprise approach that uses weighted matching scores to evaluate request-hardware compatibility, considering base model presence, adapter cache status, and queue length, with LRU-based adapter caching.
- **Proactive**: Adapted from LLM serving strategies [11, 12], this approach predicts future workloads based on historical patterns and preloads base models and frequently used adapters to reduce loading latency.

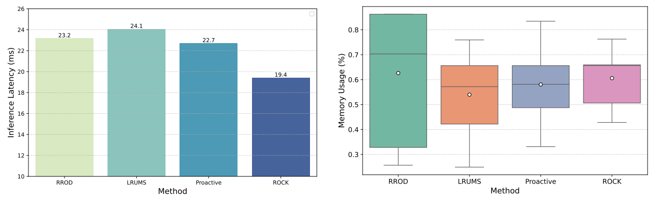
6.2 Overall End-to-End Performance

We compared Rock with baseline methods under preprocessed workloads. Fig. 10 shows the average and 99th percentile response latency. Across task types, Rock reduces average latency by 25.4%, 26.1%, and 16.6% compared to RROD, LRUMS, and Proactive respectively (Fig. 10a). The most significant improvement—41.8% reduction—appears in INPAINTING tasks compared to LRUMS, indicating that *traditional LRU caching struggles with infrequently accessed LoRA adapters*.



(a) Average Response Latency (b) P99 Response Latency

Figure 10: End-to-End Performance Comparison. (a) Average response latency (b) 99th percentile latency.



(a) Average inference latency (b) Memory Usage

Figure 11: Performance Comparison of Rock and Baselines. (a) Average inference latency under the TXT_2_IMG workload type. (b) Comparison of memory usage across methods.

Intuitively, this happens because inpainting tasks require specialized adapters that may be repeatedly evicted and reloaded under simple LRU policies, creating a "thrashing" effect that wastes computational resources.

Fig. 10b shows the 99th percentile latency, where Rock maintains values under 80s while other strategies approach 100s. This demonstrates that Rock effectively reduces tail latency, providing more stable service quality. At a fundamental level, this stability stems from Rock's ability to anticipate adapter needs and intelligently place them across the storage hierarchy, avoiding the sudden performance cliffs that occur in other systems when adapter cache misses cascade during high-demand periods. These results confirm Rock outperforms existing strategies in both average and tail latency, validating its effectiveness for LoRA-based generative tasks.

6.3 Benefit from LoRA Orchestration

We further evaluate the performance of Rock under only the TXT_2_IMG workload type, as the other two types require downloading and processing input images, which introduces variable network-dependent latency that could obscure the core system benefits. Fig. 11a shows that Rock achieves significant reductions in average inference latency: 16.8% compared to RROD, 19.5% compared to LRUMS, and 14.5% compared to the Proactive strategy.

These performance improvements stem from three key mechanisms in our LoRA orchestration: (1) the three-tier architecture effectively decouples the rigid hardware-adapter dependencies, reducing resource contention during adapter transitions; (2) the dynamic placement strategy ensures frequently accessed adapters are stored on devices with optimal bandwidth characteristics, minimizing loading overhead; and (3) our probabilistic replica management adapts to access patterns in real-time, maintaining high availability for hot adapters while preventing memory fragmentation.

TABLE 3: Loading Latency Comparison

Methods	LoRA Update Counts	Update Latency (s)	Cache Hit Rate (%)
RROD	61	6.82	0
LRUMS	45	5.78	39.2
Proactive	25	5.48	66.2
ROCK	**14**	**4.81**	**84.1**

Note: The table shows LoRA update statistics across different methods. ROCK achieves the lowest number of adapter updates and update latency while maintaining the highest cache hit rate.

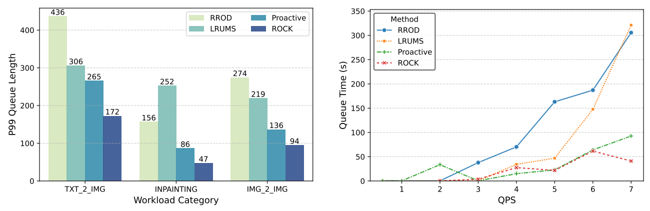
Memory usage statistics in Fig. 11b further validate our approach's efficiency. The smaller box height for Rock indicates remarkably stable memory utilization across different workloads, demonstrating that our orchestration effectively mitigates the memory volatility common in generative tasks. While achieving comparable high absolute memory utilization to the Proactive strategy, Rock does so without requiring extensive prediction models, making it more robust to unexpected workload shifts. This combination of high and stable memory utilization directly translates to improved inference throughput, as the system spends less time in memory management operations and more time on productive computation.

Table 3 shows ROCK's superior LoRA adapter management efficiency compared to other approaches. ROCK reduces adapter update frequency by 77% versus RROD and 44% versus Proactive methods, while decreasing update latency by 29.5% and 12.2% respectively. Most importantly, ROCK achieves an 84.1% cache hit rate—more than double that of LRUMS and 27% higher than Proactive. These results demonstrate the effectiveness of ROCK's multilevel orchestration framework and dynamic placement strategy in optimizing adapter placement and reducing resource contention.

6.4 Effectiveness of Dynamic Heterogeneous Queue

We analyzed queue performance across all methods. Fig. 12a shows that Rock maintains significantly shorter queue lengths compared to other approaches, demonstrating its efficiency through dynamic bipartite graph matching and Top-K optimal load balancing. This advantage is particularly evident in heterogeneous clusters, where devices vary in computing power, memory capacity, and I/O bandwidth. The dynamic heterogeneous queue effectively leverages these hardware differences to reduce waiting time and improve resource utilization.

Fig. 12b shows average queue times under varying request loads. At low QPS (<4), most methods except RROD maintain acceptable queue times. However, as traffic intensifies, LRUMS experiences dramatic increases in queue time, while Rock maintains relatively stable performance—even under high load.



(a) 90th queue length (b) Average queue time

Figure 12: 90th queue length and queue time for different method in different tasks and request distribution. (a) 99th queue length of each method under different tasks. (b) Average queue time of each method under different request distribution.

This stability stems from *intelligent workload distribution that matches task characteristics to appropriate hardware resources*, allowing the system to handle traffic spikes more effectively than traditional approaches that treat all tasks as homogeneous.

7. Related Work

In recent years, large language model (LLM) serving systems [34–42] have advanced significantly, with technologies like vLLM [43], DeepSpeed-Inference [44], LoongServe [45] and DistServe [46] improving efficiency through innovations in paged memory management and continuous batching [47]. However, the inference ecosystem for multimodal models remains fragmented, lacking the unified optimization approaches seen in LLM serving.

While LLM-focused adapter serving techniques—such as DLora's proactive loading [11], Punica's migration strategies [9], S-LoRA's efficient scheduling [10], and CaraServe's CPU-GPU coordination [12]—provide valuable insights, they inadequately address the unique challenges of multimodal systems. Specifically, these approaches fail to account for the complex topological dependencies between components like CLIP and UNet, often leading to resource contention during dynamic scheduling [16, 17, 27]. Additionally, techniques like continuous batching and request interruption recovery require substantial adaptation for long-running image generation tasks, where effective multi-level cache coordination mechanisms remain notably absent [48, 49].

Cloud provider solutions [50, 51] primarily optimize single-machine performance through hardware acceleration and memory caching, but lack systematic support for the heterogeneous characteristics of multimodal workloads and the dynamic nature of adapter management at scale. These approaches typically focus on static resource allocation and simplified caching strategies that fail to address the complex interplay between diverse adapter sizes, varying iteration requirements, and fluctuating request patterns. ROCK addresses these limitations with its specialized three-layer orchestration framework, which decouples hardware resources from adapter management while providing fine-grained scheduling tailored specifically for the unique challenges of multimodal model serving with thousands of LoRA adapters.

8. Limitations and Future Work

Despite ROCK's significant improvements in adapter management and resource utilization, our work has limitations in both middleware-level metrics and GPU hardware-level resource monitoring (memory bandwidth, compute saturation, kernel patterns). Future work will address these gaps through a comprehensive benchmark dataset integrating workload characteristics, middleware performance, and system resource utilization, enabling deeper analysis of request-hardware relationships and revealing new optimization opportunities for multimodal serving systems.

9. Conclusion

In this paper, we presented ROCK, a system for efficiently serving thousands of LoRA adapters in cloud environments through dynamic heterogeneous queues and intelligent orchestration. Our analysis of real production workloads revealed critical challenges in current serving systems, including request burstiness, adapter management inefficiencies, and resource mismatch. Experimental results demonstrate that ROCK significantly reduces adapter loading latency, increases memory utilization, and improves system throughput.

10. Acknowledgement

This work is supported by the National Natural Science Foundation of China (No. 92267105), Guangdong Basic and Applied Basic Research Foundation (No. 2023B1515130002), Guangdong Special Support Plan (No. 2021TQ06X990), Shenzhen Basic Research Program (No. JCYJ20220818101610023, KJZD20230923113800001) and Alibaba Innovative Research (AIR) Project.

References

[1] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen, "Lora: Low-rank adaptation of large language models," Oct. 2021.

[2] Rinon Gal, Yuval Alaluf, Yuval Atzmon, Or Patashnik, Amit H Bermano, Gal Chechik, and Daniel Cohen-Or, "An image is worth one word: Personalizing text-to-image generation using textual inversion," *arXiv preprint arXiv:2208.01618*, 2022.

[3] Nataniel Ruiz, Yuanzhen Li, Varun Jampani, Yael Pritch, Michael Rubinstein, and Kfir Aberman, "Dreambooth: Fine tuning text-to-image diffusion models for subject-driven generation," *arXiv preprint arXiv:2208.12242*, 2022.

[4] Guanqun Wang, Jiaming Liu, Chenxuan Li, Junpeng Ma, Yuan Zhang, Xinyu Wei, Kevin Zhang, Maurice Chong, Ray Zhang, Yijiang Liu, and Shanghang Zhang, "Cloud-device collaborative learning for multimodal large language models," 2023.

[5] Ming Zhong, Yelong Shen, Shuohang Wang, Yadong Lu, Yizhu Jiao, Siru Ouyang, Donghan Yu, Jiawei Han, and Weizhu Chen, "Multi-lora composition for image generation," 2024.

[6] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan, "Peft: State-of-the-art parameter-efficient fine-tuning methods," https://github.com/huggingface/peft, 2022.

[7] Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Yang You, Guiming Xie, Xuejian Gong, and Kunlong Zhou, "Train small, infer large: Memory-efficient lora training for large language models," 2025.

[8] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer, "Qlora: efficient finetuning of quantized llms," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2023, NIPS '23, Curran Associates Inc.

[9] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy, "Punica: Multi-tenant lora serving," 2023.

[10] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica, "S-lora: Serving thousands of concurrent lora adapters," *arXiv preprint arXiv:2311.03285*, 2023.

[11] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin, "dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, Santa Clara, CA, July 2024, pp. 911–927, USENIX Association.

[12] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang, "Caraserve: Cpu-assisted and rank-aware lora serving for generative llm inference," 2024.

[13] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, et al., "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.

[14] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, 2020, pp. 481–498.

[15] Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ramachandran Ramjee, and Rodrigo Fonseca, "Modserve: Scalable and resource-efficient large multimodal model serving," 2025.

[16] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and UC Berkeley, "Shepherd: Serving dnns in the wild," .

[17] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, JosephE. Gonzalez, and Ion Stoica, "Alpaserve: Statistical multiplexing with model parallelism for deep learning serving," Feb 2023.

[18] Liang Mi, Weijun Wang, Wenming Tu, Qingfeng He, Rui Kong, Xinyu Fang, Yazhu Dong, Yikang Zhang, et al., "Empower vision applications with lora lmm," in *Proc. Twent. Eur. Conf. Comput. Syst. EuroSys 2025 Rotterdam Neth. 30 March 2025 - 3 April 2025*. 2025, pp. 261–277, ACM.

[19] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon, "Envpipe: Performance-preserving dnn training framework for saving energy," in *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, 2023, pp. 851–864.

[20] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun, "Orca: A distributed serving system for transformer-based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[21] Yikai Zhao, Wenrui Liu, Fenghao Dong, Tong Yang, Yuanpeng Li, Kaicheng Yang, Zirui Liu, Zhengyi Jia, et al., "P4lru: Towards an lru cache entirely in programmable data plane," in *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, 2023, pp. 967–980.

[22] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka, "On-demand container loading in aws lambda," in *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, 2023, pp. 315–328.

[23] Yikai Zhao, Wenrui Liu, Fenghao Dong, Tong Yang, Yuanpeng Li, Kaicheng Yang, Zirui Liu, Zhengyi Jia, et al., "P4lru: Towards an lru cache entirely in programmable data plane," in *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, 2023, pp. 967–980.

[24] "B4: Experience with a globally-deployed software defined wan: Acm sigcomm computer communication review: Vol 43, no 4," https://dl.acm.org/doi/10.1145/2534169.2486019.

[25] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, 2022.

[26] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu, "The power of prediction: Microservice auto scaling via workload learning," in *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, 2022, pp. 355–369.

[27] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer, "High-resolution image synthesis with latent diffusion models," 2022.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017.

[29] Quoc Phong Nguyen, Sebastian Tay, Bryan Kian Hsiang Low, and Patrick Jaillet, "Top-k ranking bayesian optimization," 2020.

[30] Cinzia Viroli and Geoffrey J. McLachlan, "Deep gaussian mixture models," 2017.

[31] João M. Pereira, Joe Kileel, and Tamara G. Kolda, "Tensor moments of gaussian mixture models: Theory and applications," 2022.

[32] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach, "Sdxl: Improving latent diffusion models for high-resolution image synthesis," 2023.

[33] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 10684–10695.

[34] Yanying Lin, Yanbo Li, Shijie Peng, Yingfei Tang, Shutian Luo, Haiying Shen, Chengzhong Xu, and Kejiang Ye, "QUART: Latency-Aware FaaS System for Pipelining Large Model Inference," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1–12.

[35] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia, "SpotServe: Serving Generative Large Language Models on Preemptible Instances," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, La Jolla CA USA, Apr. 2024, ASPLOS '24, pp. 1112–1127, ACM.

[36] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han, "QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving," May 2024.

[37] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo, "Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention," in *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang, Eds. 2024, pp. 111–126, USENIX Association.

[38] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang, "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion," in *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. 2025, pp. 94–109, ACM.

[39] Lingfan Yu, Jinkun Lin, and Jinyang Li, "Stateful Large Language Model Serving with Pensieve," in *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. 2025, pp. 144–158, ACM.

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

[41] Shiwei Gao, Youmin Chen, and Jiwu Shu, "Fast State Restoration in LLM Serving with HCache," in *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. 2025, pp. 128–143, ACM.

[42] Alind Khare, Dhruv Garg, Sukrit Kalra, Snigdha Grandhi, Ion Stoica, and Alexey Tumanov, "SuperServe: Fine-Grained Inference Serving for Unpredictable Workloads," in *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*, Theophilus A. Benson and Radhika Niranjan Mysore, Eds. 2025, pp. 739–758, USENIX Association.

[43] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[44] Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song, "Fp6-llm: Efficiently serving large language models through fp6-centric algorithm-system co-design," 2024.

[45] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin, "Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism," 2024.

[46] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, Santa Clara, CA, July 2024, pp. 193–210, USENIX Association.

[47] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA, July 2022, pp. 521–538, USENIX Association.

[48] Jie Peng, Zhang Cao, Huaizhi Qu, Zhengyu Zhang, Chang Guo, Yanyong Zhang, Zhichao Cao, and Tianlong Chen, "Harnessing your dram and ssd for sustainable and accessible llm inference with mixed-precision and multi-level caching," 2024.

[49] Juncheng Yang, Yao Yue, and Rashmi Vinayak, "Segcache: A memory-efficient and scalable in-memory key-value cache for small objects," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 503–518.

[50] "Aws inferentia," https://aws.amazon.com/cn/ai/machine-learning/inferentia/.

[51] "Ai model inference service: An overview," https://www.alibabacloud.com/help/en/machine-learning-platform-for-ai/latest/pai-blade-and-inference-optimization-agile-edition.