

ZSERVE: An Adaptive Framework for Optimizing Large Language Models Serving in Heterogeneous Infrastructures

 <https://zserve.llmsys.net>

Taosong Fang^{1,2}, Zhengzhao Ma^{1,2}, Jiawei Chen^{1,2}, Yushen Yang^{1,2}, Kaijie Zhou¹,
Jia Zheng¹, Xianpei Han¹, Le Sun¹, Hongyu Lin¹, Yaojie Lu^{1,*}

¹Chinese Information Processing Laboratory, Institute of Software,
Chinese Academy of Sciences

²University of Chinese Academy of Sciences

{fangtaosong2022, mazhengzhao2024, chenjiawei2024}@iscas.ac.cn

{kaijie, zhengjia, xianpei, sunle, hongyu, luyaojie}@iscas.ac.cn

Abstract

We introduce ZSERVE, an adaptive framework designed to optimize the deployment of large language models (LLMs) in production environments. ZSERVE tackles critical challenges in LLM serving with three key innovations: (1) **Resource-aware Parameter Mining**, which automatically identifies optimal hardware configurations for various accelerators, eliminating the need for manual tuning while maximizing performance; (2) **Workload-aware Dynamic Reconfiguration**, which seamlessly switches between SLO-centric and throughput-centric serving modes based on real-time workload characteristics without service interruption; and (3) **A Unified Monitoring and Scaling Interface**, which simplifies deployment management across heterogeneous infrastructures. In contrast to static frameworks like RayServe and Xinference, ZSERVE introduces dynamic runtime adaptation, leveraging novel queue handoff and memory-efficient state migration to switch serving modes without dropping in-flight requests. Experimental results with LLMs from 7B to 70B parameters show that ZSERVE achieves up to 45% higher throughput during peak periods while preserving latency targets during moderate traffic, demonstrating consistent gains across model scales and hardware configurations.

1 Introduction

Large Language Models (LLMs) have achieved remarkable success across tasks ranging from content generation and code completion to complex reasoning tasks (Brown et al., 2020; Touvron et al., 2023). As organizations increasingly deploy LLMs in production environments, the need for robust infrastructure stacks has become evident (Qin et al., 2025). As shown in Figure 2, the modern LLM deployment stack typically follows a top-down architecture: application interfaces and APIs at the

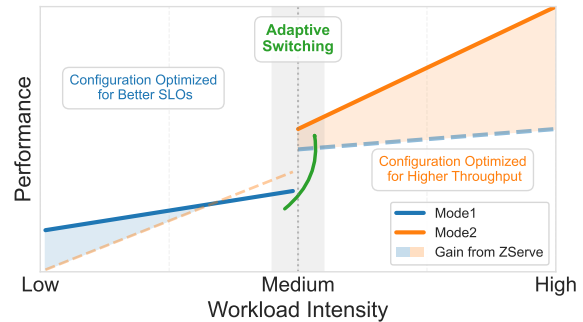


Figure 1: ZSERVE adapts between SLO-centric mode and throughput-centric mode as workload intensity changes.

highest level, serving frameworks that handle request management, inference engines that optimize computation, and hardware accelerators at the foundation (Zheng et al., 2024; Kwon et al., 2023). Each layer addresses a distinct aspect of the serving process, ranging from user-level functionality to computational optimization.

Despite substantial progress across each layer, current LLM serving solutions have a significant limitation: the absence of adaptive middleware capable of dynamically adjusting serving parameters during runtime (Li et al., 2023). Currently, operators must choose static configurations optimized either for latency or throughput when deploying models. This inflexibility restricts adaptability, especially under variable workloads characterized by fluctuating request concurrency and diverse input-output sequence lengths.

This limitation becomes particularly problematic when considering Service Level Objectives (SLOs)—quantifiable metrics that define expected service performance such as response time or availability—which are critical for user-facing LLM applications. Organizations typically establish SLOs such as maximum acceptable latency for the first token generation or consistent token production rate

*Corresponding author

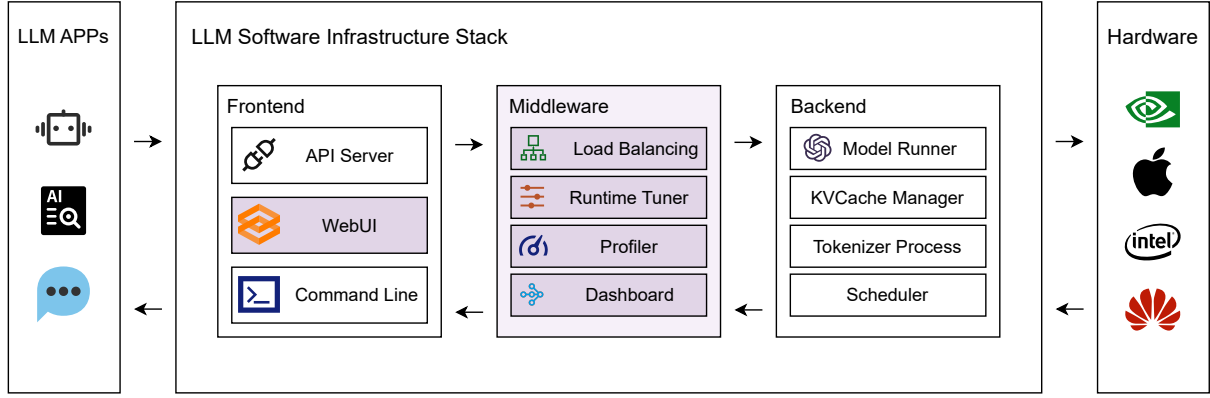


Figure 2: The LLM software stack follows a top-down architecture, with application interfaces at the top and hardware acceleration at the bottom. The highlighted section illustrates the role of ZSERVE as an adaptive middleware layer that bridges the gap between static serving frameworks and dynamic, variable workload patterns.

to ensure satisfactory user experiences (Patel et al., 2024; Zhong et al., 2024). However, traditional serving frameworks force system architects into suboptimal compromises: either provision multiple specialized instances for different workload patterns, increasing operational complexity and resource costs, or accept performance degradation when actual usage patterns diverge from the initial configuration assumptions (Liu et al., 2024; Rasley et al., 2020; Fang et al., 2021). Neither approach efficiently utilizes the underlying computational resources or provides consistent service quality across varying demands.

To address these challenges, we present ZSERVE, an adaptive middleware layer that bridges static serving frameworks and dynamic workload requirements through three key innovations. First, our resource-aware parameter mining technique automatically discovers optimal, hardware-specific configurations, eliminating manual tuning while maximizing performance across diverse accelerator-model combinations. Second, as illustrated in Figure 1, our dynamic reconfiguration module enables seamless transitions between latency-optimized (SLO-centric) and throughput-optimized serving modes based on real-time workload characteristics, without restarting the service or dropping in-flight requests. Third, ZSERVE offers a unified monitoring and scaling interface that abstracts hardware heterogeneity, enabling consistent operational workflows and metrics collection across different types of accelerators.

Our evaluation on industry-standard LLMs demonstrates that ZSERVE achieves up to 45% higher throughput during peak load periods while maintaining latency targets during moderate traffic

compared to static configurations. The system’s automated parameter mining replaces extensive manual tuning with a systematic profiling workflow, while its seamless mode transitions eliminate the need for instance restarts when adapting to changing workloads. By introducing dynamic adaptability to the LLM serving stack, ZSERVE represents a significant step toward more efficient and resilient production deployments of large language models.

2 System Design and Architecture

Optimizing LLM serving configurations presents two fundamental challenges: first, the performance landscape varies dramatically across hardware platforms, model architectures, and workload patterns; second, static configurations cannot efficiently adapt to dynamic workload fluctuations. ZSERVE addresses these challenges through a system architecture that separates configuration discovery from runtime adaptation.

ZSERVE comprises four integrated components: (1) an offline profiling engine that systematically maps the performance landscape, (2) a configuration selector that initializes deployments with optimal parameters, (3) a runtime adaptation module that dynamically reconfigures serving parameters, and (4) a monitoring interface that provides visibility and control over the system’s behavior.

2.1 Resource-Aware Parameter Mining

Manual discovery of optimal serving parameters for each hardware configuration is prohibitively time-consuming and often results in suboptimal performance. ZSERVE employs a systematic parameter mining process that automatically identifies hardware-specific performance profiles, evalu-

Algorithm 1 Offline Profiling

Input: Dataset or data distribution $\mathcal{D}$, Hardware environment $\mathcal{H}$, Workload patterns $\mathcal{W}$

Output: Configuration library $\mathcal{L}$

$\mathcal{L} \leftarrow \{\}$

Load $\mathcal{D}$ or generate synthetic data for $w \in \mathcal{W}$.

for parallel configuration (TP, PP, PD) **do**

for chunk size $cs \in \{512, 768, 1024, \dots\}$ **do**

for query per second $q \in \{1, 2, 4, \dots\}$ **do**

 Configuration $\mathcal{C} \leftarrow (\text{TP}, \text{PP}, \text{PD}, cs, q)$.

 Deploy $\mathcal{C}$ on $\mathcal{H}$.

 Execute workload pattern $w_{\mathcal{C}}$ with $\mathcal{C}$.

 Record TTFT, TPOT, Throughput.

end for

end for

end for

for $w \in \mathcal{W}$ **do**

for $\mathcal{C}$ s.t. $w_{\mathcal{C}} = w$ **do**

 Evaluate performance for $\mathcal{C}$.

end for

 Identify the best-performing configurations under each serving objective.

 Append the selected configurations to $\mathcal{L}$.

end for

return $\mathcal{L}$

ating LLM serving behavior using four critical metrics: Time to First Token (TTFT) measures initial response latency, Time Per Output Token (TPOT) captures generation speed, throughput measures overall system capacity, and SLO attainment rate quantifies the percentage of requests meeting pre-defined service level objectives.

Algorithm 1 outlines our performance profiling approach. For each hardware environment, we systematically explore the parameter space, varying parallelism strategies such as Tensor Parallelism (TP), Pipeline Parallelism (PP), and PD-Disaggregation (Zhong et al., 2024) as well as batch configurations and request rates. The algorithm first characterizes workloads into canonical patterns—short input/short output (SISO), short input/long output (SILO), long input/short output (LISO), and long input/long output (LILO), to ensure optimizations reflect realistic usage scenarios.

The profiles are categorized by workload pattern and stored in a configuration library that serves as the foundation for intelligent runtime decisions. This automated process replaces extensive manual tuning with a systematic, reproducible profiling workflow.

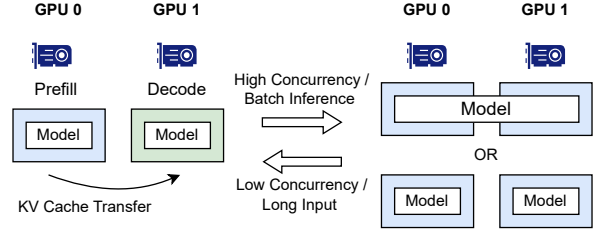


Figure 3: An example of dynamic reconfiguration in ZSERVE. Under low-concurrency workloads (left), the system can employ a PD-Disaggregation mode. As workload patterns shift, the profiler determines that a hybrid deployment is optimal (right).

2.2 Automatic Configuration Selection and Dynamic Reconfiguration

Traditional LLM deployments rely on static parameter selection, which often comes with obscured performance implications. ZSERVE overcomes this limitation through automated initialization and continuous adaptation. During deployment, the configuration selector detects the underlying hardware and retrieves relevant performance profiles from the configuration library. It then applies heuristic matching to align stored profiles with current deployment scenarios. Only high-level objectives (e.g., latency sensitivity or throughput maximization) need to be specified in this process. ZSERVE automatically derives optimal parameters based on objectives and hardware characteristics.

Beyond initial configuration, ZSERVE’s runtime adaptation module continuously monitors performance metrics to detect workload shifts. When a deviation is observed, the system assesses potential reconfigurations to enhance performance. As illustrated in Figure 3, ZSERVE can switch from a PD-Disaggregation mode to a setup with chunked prefill (Holmes et al., 2024) and tensor parallelism to better accommodate the current load.

A parallelism reconfiguration (e.g., switching from TP=4 to DP=4) involves four coordinated phases that together enable live, non-disruptive transitions:

(1) **Weight Resharding.** Changing the tensor parallelism degree requires redistributing model weights across GPUs. For example, transitioning from TP=4 (each GPU holds $\frac{1}{4}$ of the weight shards) to DP=4 (each GPU holds a full replica) requires an all-gather operation to reconstruct complete weight tensors on each device. Conversely, scaling up TP involves scatter operations to partition weights. On NVLink-connected GPUs, this

transfer completes in sub-second time even for models with tens of billions of parameters.

(2) Execution Graph Re-instantiation. Accelerator-specific execution graphs (e.g., CUDA Graphs on NVIDIA GPUs, ACL Graphs on Huawei Ascend NPUs) bind to specific device memory addresses and tensor shapes, and thus cannot be reused across different parallelism configurations. ZSERVE pre-captures graph templates for each target configuration at startup; during a transition, a new instance is instantiated from the corresponding template ($< 1s$).

(3) Request Migration. When a reconfiguration changes the instance count from m to $2^n \cdot m$, in-flight requests are redistributed deterministically. For scaling up ($n > 0$), all requests and their associated metadata on instance p are distributed across new instances in the range $[p \cdot 2^n, (p+1) \cdot 2^n - 1]$ according to a load-balancing policy. For scaling down ($n < 0$), requests on instance p are consolidated onto instance $\lfloor p \cdot 2^n \rfloor$. This deterministic mapping avoids request duplication or loss during transitions.

(4) KV Cache Rebuilding. Under tensor parallelism, the KV cache is sharded along the `num_kv_heads` dimension across GPUs. A change in TP degree invalidates the existing shard layout, requiring KV cache reconstruction. Rather than recomputing KV states from the original context (which would incur full prefill cost), ZSERVE performs direct cross-device memory copies to re-assemble KV cache blocks into the target shard configuration. This approach preserves all cached states while avoiding redundant computation.

Together, these four phases enable ZSERVE to complete a full parallelism reconfiguration (e.g., $TP=4 \rightarrow DP=4$ for a 14B-parameter model on $4 \times A100-80GB$ with NVLink) within approximately 5 seconds under moderate load (~ 20 concurrent requests, average sequence length 2048). By dynamically switching between SLO-centric configurations (optimized for consistent response times) and throughput-centric configurations (maximizing processing capacity), a single deployment can efficiently serve heterogeneous workloads that would otherwise require distinct specialized instances.

2.3 Unified Monitoring Interface

ZSERVE provides a unified monitoring interface that integrates programmatic APIs, a command-line interface (CLI; Appendix A), and an intuitive

Web-based dashboard (Figure 4). This interface collects and visualizes key metrics (e.g., throughput, GPU utilization) and adaptation events in real-time via WebSocket connections. Beyond passive monitoring, it supports active management across three core modules: (1) **Model Repository Management** for registering and versioning models with hardware compatibility metadata; (2) **Deployment Configuration** for fine-grained control over execution parameters (e.g., batch size, KV cache allocation); and (3) **Runtime Monitoring** for tracking performance distributions and triggering scaling actions. This unified design significantly reduces operational complexity in dynamic multi-tenant environments.

3 Implementation

ZSERVE is implemented as an integration layer for popular LLM inference engines, currently supporting vLLM and SGLang on NVIDIA GPUs and MindIE on Huawei Ascend accelerators, while maintaining a consistent interface for application developers. We developed a RESTful API server that exposes unified endpoints for model management, inference requests, and performance monitoring. Most notably, it supports runtime weight redistribution without instance restarts, which is essential for ZSERVE’s live reconfiguration between serving modes. Figure 5 shows how ZSERVE integrates with user-facing interfaces and serving engines.

4 Evaluation

Setup. To comprehensively evaluate ZSERVE’s adaptive capabilities, we conducted experiments across multiple model scales and hardware configurations. Our evaluation setup is as follows: **(1) Models.** We use Qwen2.5-7B, Qwen2.5-14B, and LLaMA3.1-70B to assess scalability across different architectures and parameter counts. **(2) Hardware configurations.** We deploy the 7B and 14B models on a single node with $2 \times A100-80GB$ GPUs, and the 70B model on a single node with $8 \times H800-80GB$ GPUs. **(3) Baseline comparison.** We use vLLM services deployed through Xinference with default and optimized static configurations as baselines.

Baselines. Our primary baseline consists of vLLM model services deployed through Xinference with static configurations. We established two

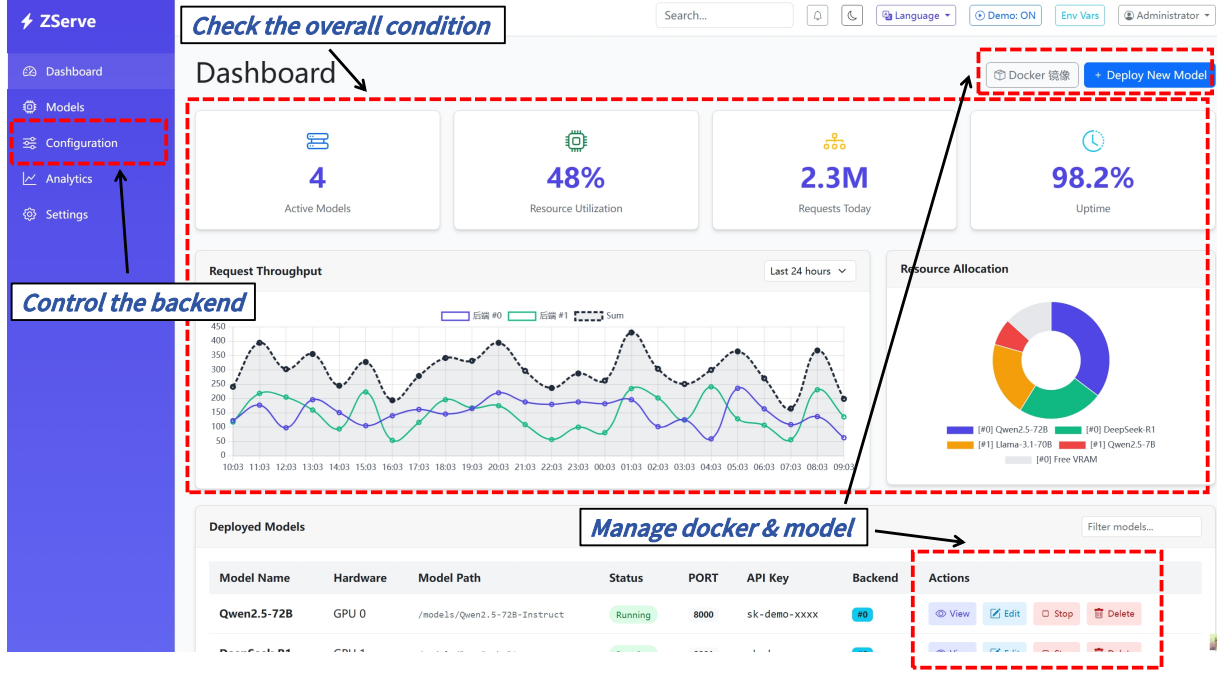


Figure 4: Web-based monitoring and scaling interface of ZSERVE, illustrating its core functional modules and their integration with the underlying middleware components.

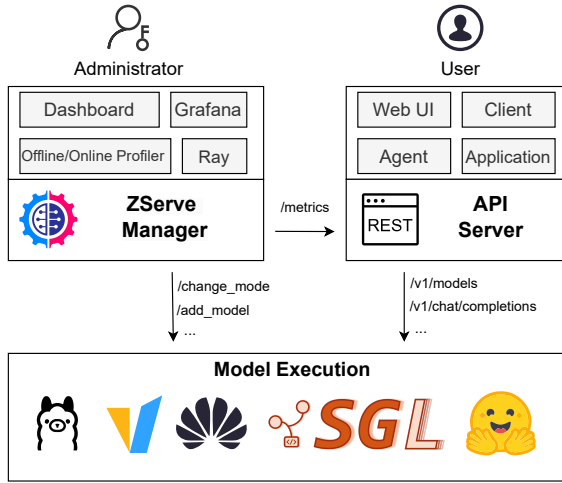


Figure 5: ZSERVE’s integration with user-facing interfaces and serving engines.

baseline variants: (1) **PD-Disaggregation configuration** optimized for latency-sensitive workloads with predictable response times; (2) **Chunked Prefill configuration** optimized for throughput-intensive batch processing scenarios. Both baselines utilize identical hardware resources and are evaluated under the same synthetic workload patterns to ensure fair comparison.

Service Level Objectives. We use the following SLO targets: Qwen2.5-7B (TTFT ≤ 100 ms, TPOT

≤ 30 ms), Qwen2.5-14B (TTFT ≤ 200 ms, TPOT ≤ 50 ms), and LLaMA3.1-70B (TTFT ≤ 150 ms, TPOT ≤ 50 ms in TP=2, TTFT ≤ 80 ms, TPOT ≤ 30 ms in TP=4).

Results. Figure 6 presents our comprehensive performance evaluation. At low query rates (QPS 1-4), where SLO compliance is paramount, ZSERVE operates in SLO-centric mode and achieves SLO-attainment rates comparable to the PD-Disaggregation baseline while significantly outperforming the Chunked Prefill configuration. This demonstrates ZSERVE’s ability to maintain service quality during moderate traffic periods.

As workload intensity increases (QPS 16-64), where strict SLO requirements become challenging to maintain, ZSERVE transitions to throughput-centric mode. This adaptive reconfiguration enables ZSERVE to achieve up to 45% higher normalized throughput compared to static PD-Disaggregation deployments, while maintaining competitive performance against the throughput-optimized Chunked Prefill baseline. Notably, this performance gain is achieved without manual intervention, instance restarts, or request loss, highlighting the practical value of dynamic adaptation in production environments where workload patterns fluctuate unpredictably.

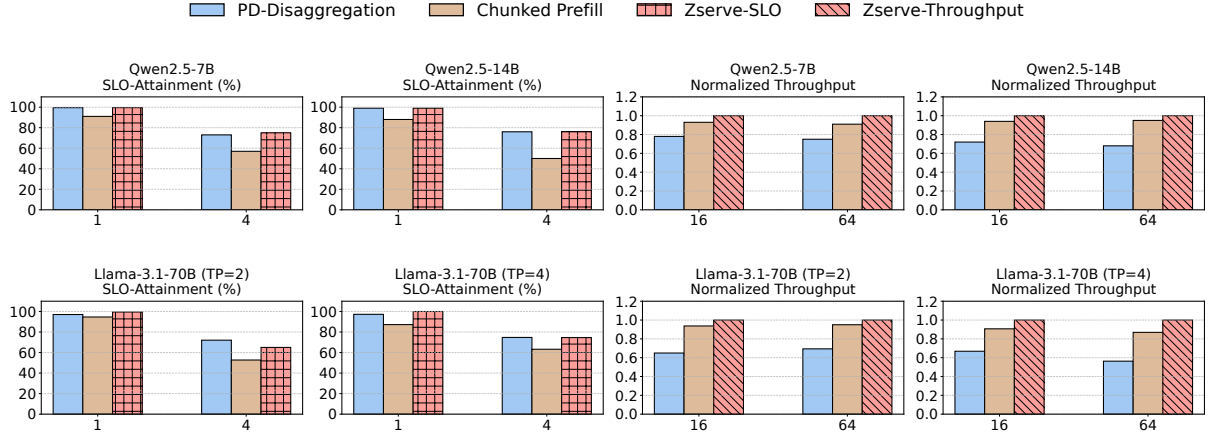


Figure 6: Performance comparison of ZSERVE against baseline methods on Qwen2.5 models (7B and 14B parameters) and LLaMA3.1-70B under different tensor parallelism configurations (TP=2 and TP=4).

5 Related Work

Recent advancements in LLM serving have emerged across model execution, operator optimization, and deployment frameworks. On the model execution side, specialized inference engines such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) employ continuous batching and efficient scheduling mechanisms that substantially improve memory efficiency and hardware utilization. In parallel, low-level operator optimizations such as FlashAttention (Dao et al., 2022; Dao, 2023) have focused on improving the core computation efficiency of transformer architectures. By leveraging I/O-aware algorithms and accelerator-specific memory patterns, these methods reduce runtime latency and improve throughput in GPU-heavy workloads.

To manage deployment complexity, several unified serving frameworks have been proposed. For instance, Xinference (Lu et al., 2024) and RayServe (Moritz et al., 2018) provide abstractions for heterogeneous hardware, along with standard APIs and resource management capabilities. However, these systems primarily focus on horizontal scaling (adding/removing instances) rather than vertical adaptation (modifying serving configurations within instances). This limitation becomes particularly evident under workloads with rapid fluctuations where instance startup latency makes horizontal scaling ineffective.

Despite these advancements, a common limitation remains: most systems require static configuration choices that cannot adapt to evolving workload patterns. This often results in a fixed trade-off between throughput-oriented batch processing

and latency-sensitive request serving. Building on the aforementioned inference engines, our work focuses on *dynamic adaptation* mechanisms by proposing a unified serving system that dynamically reconfigures serving parameters in response to observed workload patterns, eliminating the trade-off between throughput and latency optimization.

6 Conclusion

By incorporating resource-aware parameter mining, workload-aware dynamic reconfiguration, and a comprehensive monitoring and scaling interface, ZSERVE demonstrates significant improvements in throughput and resource utilization while maintaining SLO compliance. The system seamlessly transitions between SLO-centric and throughput-centric modes without restarting the service or dropping in-flight requests, adapting to changing workload patterns in real-time. This adaptive approach eliminates the traditional need to choose between optimizing for latency or throughput, allowing a single deployment to efficiently handle diverse operational scenarios. As LLM applications continue to proliferate, ZSERVE’s adaptive middleware represents an important step toward making industrial-scale language model deployment more accessible, efficient, and cost-effective across heterogeneous infrastructure environments.

Acknowledgments

We sincerely thank the reviewers for their insightful comments and valuable suggestions. This work was supported by the National Key R&D Program of China (2024YFC3308000), the Natu-

ral Science Foundation of China (No. 62306303, 62476265, 62272439). The authors would like to thank Huawei Ascend Cloud Ecological Development Project for the support of Ascend 910 processors.

References

- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.
- Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359.
- Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402.
- Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and 1 others. 2024. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.
- Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, and 1 others. 2023. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679.
- Xiaoxuan Liu, Cade Daniel, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Xiangxi Mo, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. 2024. Optimizing speculative decoding for serving large language models using goodput. *arXiv preprint arXiv:2406.14066*.
- Weizheng Lu, Lingfeng Xiong, Feng Zhang, Xuye Qin, and Yueguo Chen. 2024. [Xinference: Making large model serving easy](#). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 291–300, Miami, Florida, USA. Association for Computational Linguistics.
- Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, and 1 others. 2018. Ray: A distributed framework for emerging ai applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577.
- Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE.
- Ruoyu Qin, Zheming Li, Weiran He, Jiale Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. [Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot](#). In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA. USENIX Association.
- Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.
- Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and 1 others. 2024. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583.
- Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210.

A Command-line Interface

ZSERVE provides a comprehensive CLI that supports the complete serving lifecycle through intuitive commands for model management, deployment control, and runtime configuration.

Performance Profiling Before deployment, users can generate hardware-specific performance profiles to guide configuration selection:

```
zserve-cli profile --model Qwen2.5-7B
--data-type auto --output-path
./profiles
```

Server Initialization The serving infrastructure can be launched with configurable networking parameters:

```
zserve-cli start-server --port 8080
--host 0.0.0.0 --enable-dynamic-
reconf
```

Model Deployment Adding models to the serving pool supports configuration customization and dynamic reconfiguration options:

```
zserve-cli add --model Qwen2.5-7B --
config-file config.json --enable-
dynamic-reconf --backend vllm
```

Resource Management Models can be removed from the serving infrastructure when no longer needed:

```
zserve-cli remove --model Qwen2.5-7B
```

Runtime Reconfiguration Administrators can manually trigger serving mode transitions as needed:

```
zserve-cli change-mode --target-mode
throughput-centric
```

The CLI integrates intelligent load-balancing that automatically routes incoming requests to appropriate model instances based on the model name and available resources, eliminating the need for manual shard allocation or request distribution.