MLSys 2026

# GhostServe: A Lightweight Checkpointing System for Fault-tolerant LLM Serving

Shakya Jayakody*, Youpeng Zhao*†, Chinmay Nehate, Jun Wang

*Computer Systems and Data Science (CASS) Lab*

*University of Central Florida*

*: Equal Contribution

†: Project Lead

# Background



## The Emergence of Large Language Models (LLMs)

- Starts with Transformers (self-attention)
- Grows with scaling laws (GPT models)
- Explodes with ChatGPT and open-sourced models (LLaMA and DeepSeek)

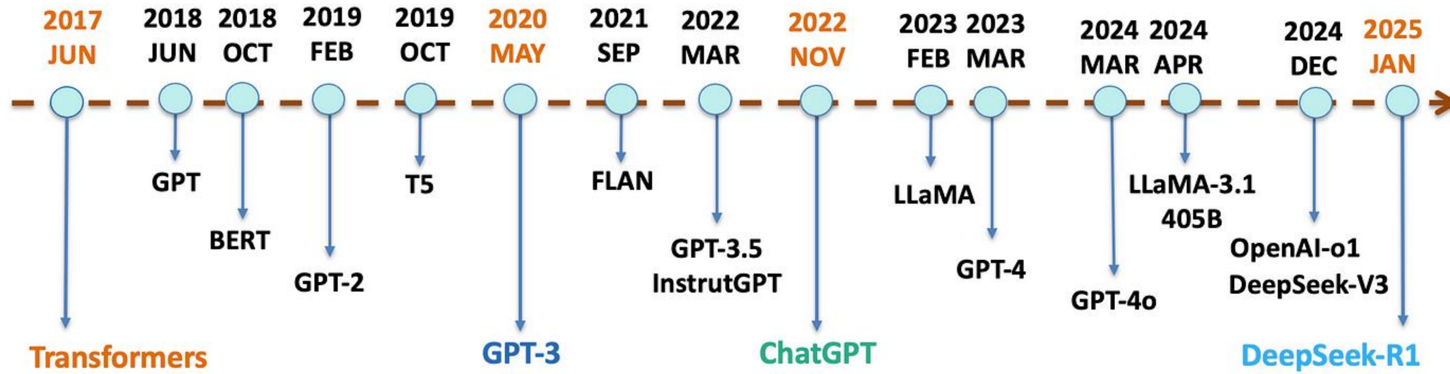



**Fig. 1: Evolution of large language models (LLMs) in recent years.**

# Background

## The Rising Demand for LLM Inference Services

- **Trend I: Long-context Tasks**
    - Frontier models with 1M+ context window (Claude/Gemini)
    - Heavy RAG and agentic workloads have become the norm

**Workload Intensity ↑**

- **Trend II: Distributed Serving**
    - Large models require multiple GPUs or nodes to deploy
    - Complex parallelism strategies (DP/TP/CP/SP/PP/EP)

**Infrastructure Scale ↑**

**Fig. 2: An inaccurate depiction of LLM serving.**

# Motivation



## The Achillies of Large-scale Systems – Fault Tolerance

- Large distributed systems are prone to hardware and software errors, where interruptions are the norm instead of exception, especially at scale
- Untimely recovery from faults can lead to potentially significant financial and operational losses

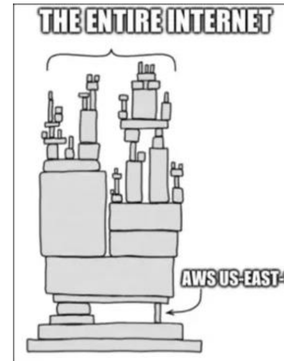



**Fig. 3: Two most recent major global outages: 2024 CrowdStrike bluescreen incidents (left) and 2025 AWS October Incident (right).**

# Motivation

MLSys 2026

## LLM serving requires specialized fault tolerance support

- The increased workload intensity and serving scale (4K → 1M)
- We need to protect the streaming KV cache, which dynamically grows with context length. For larger models, the KV cache memory footprint can amount to terabytes (TB)

Solution I: Recomputation

- No additional system overhead
- Work for shorter requests
- Longer requests might take tens of minutes

Solution II: Replication

- Easy to implement and potentially faster than recomputation
- GPU-CPU I/O traffic for normal serving operations, potentially degrading system throughput

# Method



## How do we protect the growing KV cache with minimal overhead?

## Core Idea: Erasure Coding (RAID)

- A method of data protection in which data is broken into fragments, expanded and encoded with redundant data pieces, and stored across a set of different locations or storage media.

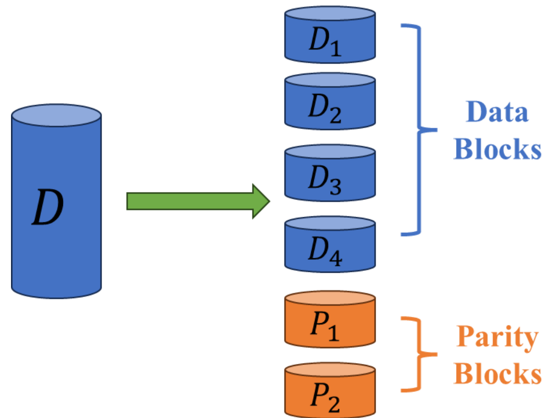



**Fig. 4: Illustration of erasure coding.**

Encoding:

$$P_1 = f_1(D_1, D_2, D_3, D_4) \qquad P_2 = f_2(D_1, D_2, D_3, D_4)$$

Reconstruction: If $D_1$ is lost/corrupted,

$$D_1 = f_1^{-1}(P_1, D_2, D_3, D_4) \qquad D_1 = f_2^{-1}(P_2, D_2, D_3, D_4)$$

Common erasure codes: XOR, RDP, Reed-Solomon

# Challenges

- How to implement GPU-based erasure coding for floating-point (FP) KV Cache → Low latency
- How to implement the erasure coding in distributed settings, such as tensor parallelism (TP) → Low memory and communication overhead
- How to incorporate the checkpointing mechanism in popular backend engine platforms, such as SGLang → High portability

MLSys 2026

## - Design I: Optimized GPU Kernels

**Challenges:**

- Format incompatibility (floating point vs. integer)
- Excessive memory access times (PyTorch)

**Techniques:**

- Lossless integer-centric reinterpretation of floating point KV cache
- Native CUDA kernels with grid-stride loops and warp-synchronous execution

| Tensor Size | Ours (CUDA) | | PyTorch | |
|---|---|---|---|---|
| | Enc. (ms) | Rec. (ms) | Enc. (ms) | Rec. (ms) |
| 2 GB | 6.72 | 5.42 | 48.63 | 23.63 |
| 4 GB | 11.74 | 10.27 | 61.41 | 45.90 |
| 8 GB | 20.17 | 17.30 | 106.58 | 91.76 |
| 16 GB | 38.71 | 32.84 | 199.67 | 182.10 |

5.7 × speedup on encoding

4.9 × speedup on reconstruction

# System Design

MLSys 2026

## - Design II: Distributed Scheduler

Challenges:

- How to perform erasure coding for distributed KV cache
  - Checkpointing granularity
  - Intra-node communication

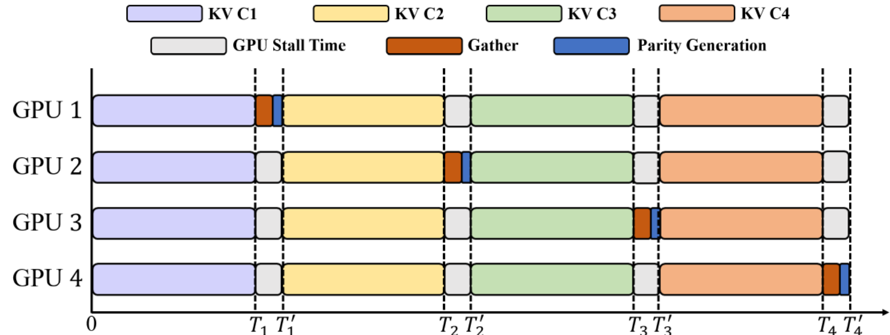



**Fig. 5: Illustration of distributed scheduler.**

Techniques:

- *Chunk-level Checkpointing*: KV cache is **append-like and write-once-read-many**
- *Gather-Encode-Offload*: View KV cache from different workers as different data shards and gather them altogether to generate parity. Parity is offloaded to host DRAM in asynchronous fashion to overlap with computation
- *Load Balancing*: rotating the encoding operation among devices to avoid straggler effects

# System Design

MLSys 2026

## - Design III: Hybrid Recovery

Challenges:

- How to efficiently recover the KV cache to resume the inference process

Techniques:

- For initial portion of the KV cache, we can use recomputation to recover
- The remaining segments of KV cache are recovered using erasure coding to reconstruct on other GPUs combined with parity from host DRAM.

Minimizes GPU idle time, overlapping computation with I/O transfer and improve recovery speed

**Algorithm 2** GhostServe Recovery Workflow.

**Initialization:** Initialize erasure coding compute unit *EC*, KV cache $KV$, chunk size $m$, input length $s$, number of prefill chunks $c = \lceil s/m \rceil$, the number of GPUs $N$, number of parity shards $K$. Assuming $k$-th GPU failed after $n$-th chunk.

1: ...
2: Failure.detect()
3: # Calculate the recompute units
4: $r =$ get_recompute_units$(s, m, N, K)$
5: **if** $r >= n$ **then**
6: # Perform recomputation
7: **for** all $i < n$ **do**
8: $KV_i^k =$ GPU$^k$.Process$(s, i)$
9: **end for**
10: **else**
11: # Perform recomputation + reconstruction
12: **for** all $i < r$ **do**
13: $KV_i^k =$ GPU$^k$.Process$(s, i)$
14: **end for**
15: # Gather remaining KV cache to $k$-th GPU
16: $KV_{1:n-r} =$ torch.dist.gather$(KV_{1:n-r}^{1:N}, \text{GPU}^k)$
17: $P_{1:n-r} \xrightarrow[\text{Async}]{\text{PCIe}} \text{GPU}^k$
18: # Perform reconstruction
19: With CUDA.Streams():
20: $KV_{1:n-r} =$ EC.reconstruct$(KV_{1:n-r}, P_{1:n-r})$
21: **end if**
22: Serving.Resume()
23: ...

# Implementation

- Backend engine: SGLang 0.5.1, PyTorch 2.8, CUDA 12.6, FlashInfer 0.3.1, NCCL 2.21.5, with 4K+ Python, 1.5K+ C++/CUDA LoCs

- Support Models: LLaMA-8B, LLaMA-70B, GPT-OSS-20B, GPT-OSS-120B, DeepSeek-R1-32B

- Kernel Optimizations:
  - Kernel Fusion: Fusing conversion, encoding and reconstruction in one pass
  - CUDA Graph: Capture the fixed size graph and replay during execution
  - CUDA Streams: Concurrent recovery for different chunks

- Code Demo: https://github.com/project-ghostserve/26mlsys-AE-GhostServe

# Evaluation

MLSys 2026

## Experimental Settings:

- **Platform**: 8 NVIDIA H200 GPUs with NVLink Gen 4, 1024GB DRAM and PCIe Gen 4
- **Baselines**: (a) Recomputation: *SGLang-Base*; (b) *Replication*: SGLang-CPU/-SSD
- **Workloads**: Medha generator (4K~1M) with varying batch sizes and Poisson arrival time
- **Metrics**: (a) Batched Inference: Prefill/decode/recovery latency; (b) Online Serving: P50/99 latency, Effective-Inference-Time-Ratio (EITR), Mean-Time-To-Recover (MTTR)
- **Fault Simulation**: Randomly inject errors to mimic device faults by flushing the memory buffers with varying failure rate

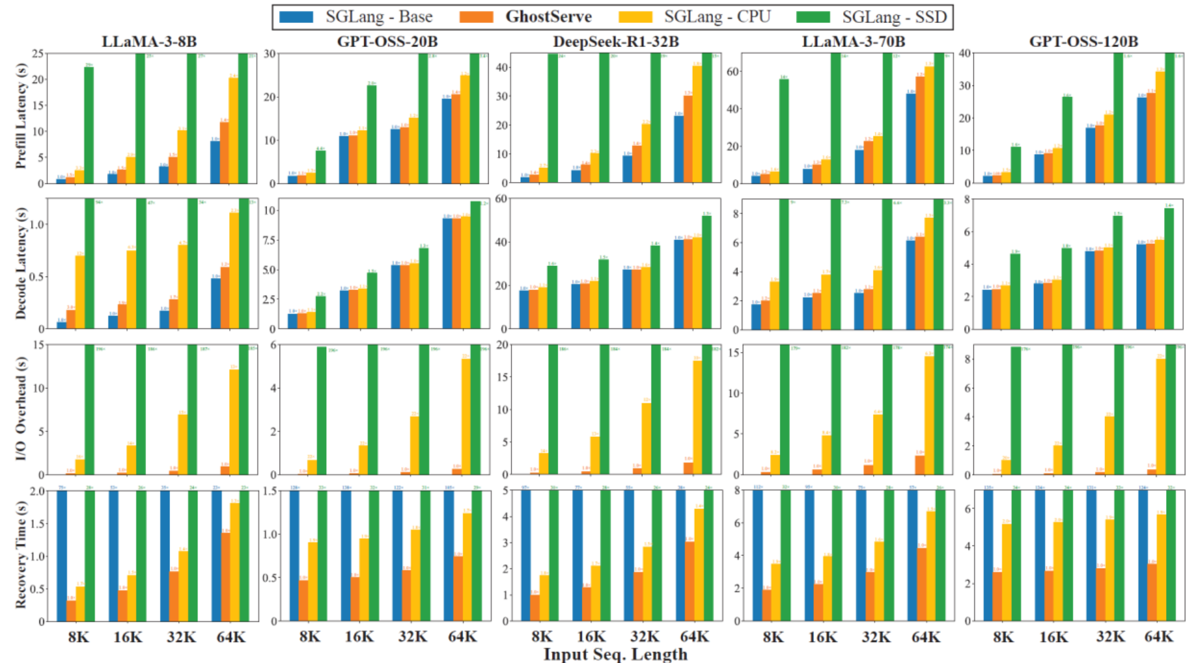
# Evaluation

MLSys 2026

## Batched Inference Results

- GhostServe consistently outperforms replication-based methods in terms of checkpointing latency overhead (up to 11x compared to CPU method)

- GhostServe achieves the lowest recovery time, especially on large models (seconds instead of minutes)

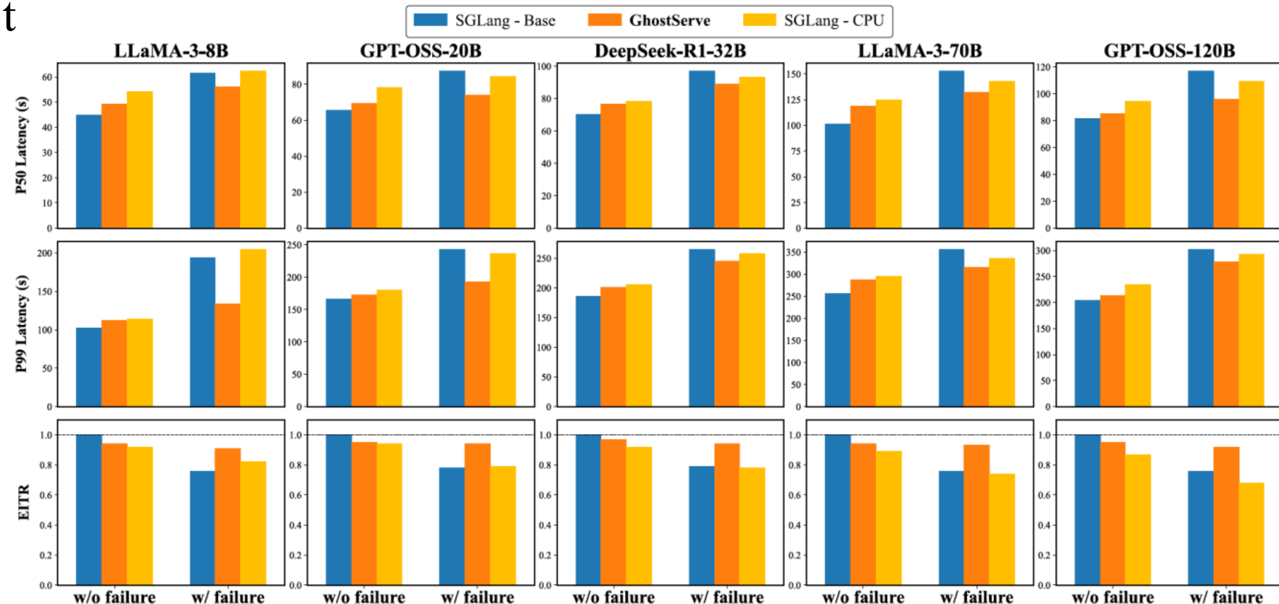


# Evaluation

MLSys 2026

## Online Serving Results

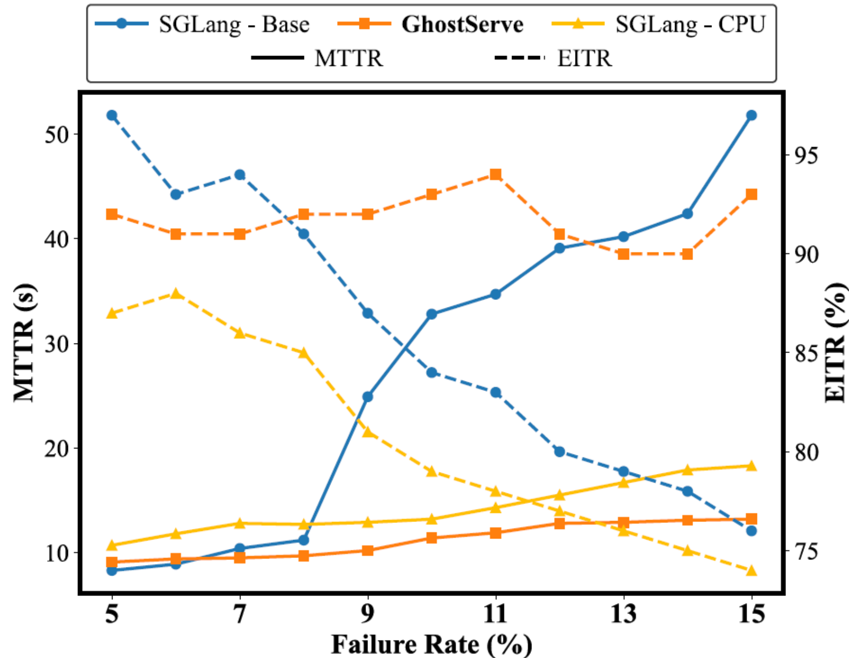
- GhostServe provides lowest overheads in failure-free scenarios (11% less)
- GhostServe achieves speedups for response latency in failure-induced cases (up to 1.2x)
- GhostServe achieves consistently high EITR (>90%)



# Evaluation

MLSys 2026

## Cost-Benefit Analysis



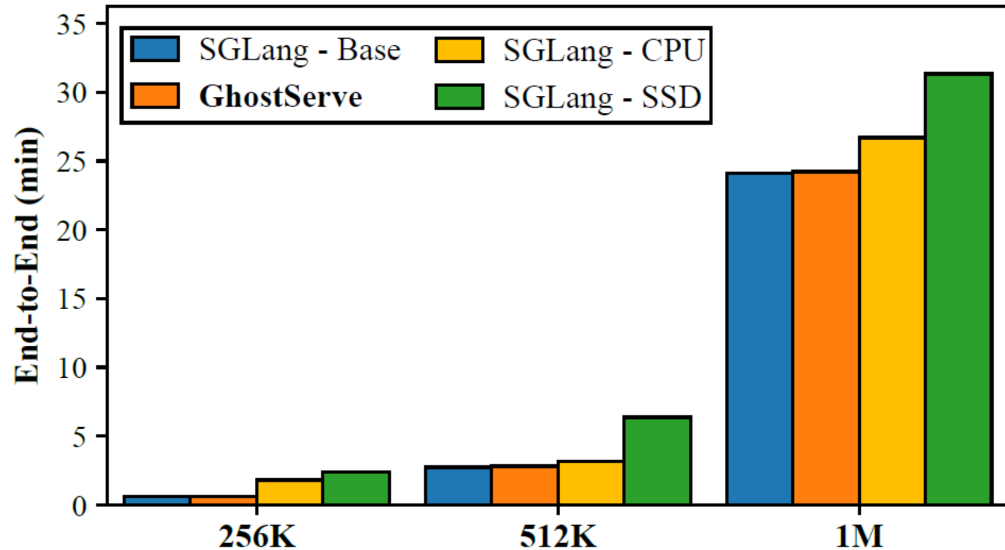

- GhostServe maintains high EITR under varying failure rate (robustness)
- GhostServe provides much lower MTTR, especially in higher failure rates (>8%)

# Evaluation

MLSys 2026

## Scaling to Million Tokens

- GhostServe achieves lowest overheads among all checkpointing methods (< 6%)
- GhostServe reduces checkpointing from minutes to seconds



# Summary

MLSys 2026

## Conclusion

- We identify the reliability challenges in LLM serving and propose a checkpointing system based on the idea of erasure coding to protect KV cache
- We design and implement highly optimized GPU kernels, and propose chunk-level checkpointing with load balancing and hybrid recovery strategy for LLM serving in distributed environments
- Evaluations demonstrate that GhostServe consistently outperforms existing methods and can significantly reduce the checkpointing overhead

## Future Work

- Extension to cross-node environments (PP/CP/DP)
- Extension to PD disaggregated scenarios
- Full-stack fault tolerance

# Acknowledgements

MLSys
2026


- We appreciate the generous support from:
  - NSF grant 1907765, 2400014, and 2426368
  - Lambda AI research grant
  - NSF ACCESS program (NCSA Delta and DeltaAI)

- Faculty and student investigators from the UCF Computer Systems and Data Science (CASS) Laboratory

- Patent Pending:
  - Wang, J. and Zhao, Y. Dynamic erasure-coded KV cache with live migration for LLM inference. U.S. Patent Application 63877653, September 2025.

# Thank you!