# HW11

Tguo67

**Question(s):**

Read the related materials, and answer the following questions:

1. Based on the paper of Llumnix, why is request rescheduling needed? What are the scenarios where request scheduling can help?
2. How does Llumnix mitigate the time latency caused by live migration?
3. How does KunServe serve requests efficiently after dropping the parameters?

Related materials:
Llumnix paper: https://www.usenix.org/system/files/osdi24-sun-biao.pdfLinks to an external site.
KunServe paper: https://ipads.se.sjtu.edu.cn/_media/publications/kunserve-eurosys26.pdfLinks to an external site.

**Answers**:

**1. Based on the paper of Llumnix, why is request rescheduling needed? What are the scenarios where request scheduling can help?**

- **The reason why request rescheduling needed:** LLM serving faces heterogeneous and unpredictable requests (varying prefill/decode lengths, unknown output lengths), which break isolation, create GPU-memory fragmentation, and make it hard to honor priorities/SLOs. Llumnix treats the cluster like an OS and enables runtime rescheduling across instances to react to these dynamics
- **Scenarios:**
  - Load balancing to reduce interference/preemptions among active batches.
  - De-fragmentation to free a contiguous block for incoming long-prefill jobs (cuts head-of-line queuing).
  - Prioritization to isolate/accelerate high-priority requests by reserving headroom.
  - Auto-scaling to quickly saturate new instances or drain terminating ones via migration.
  - These are unified via the "virtual usage" abstraction and a freeness-based load balancer

**2. How does Llumnix mitigate the time latency caused by live migration?**

The KV cache is append-only, Llumnix performs multi-stage, pipelined live migration that overlaps KV-cache copying with ongoing decoding, so the only stall is copying the *latest* block generated in the current step (near-zero, constant w.r.t. sequence length). A handshake protocol coordinates source/destination, avoids OOM during continuous batching, and commits at the end. Evaluations show "near-zero downtime" and minimal overhead to non-migrated requests.

**3. How does KunServe serve requests efficiently after dropping the parameters?**

- **Trigger:** when HBM is overloaded (bursts) and KV-centric tactics (drop/swap/migrate KV) can't free enough space fast, KunServe selectively drops replicated model parameters to instantly free large memory, then still serves all requests correctly by cooperative execution on GPUs that retain full parameter copies.

- Cooperative pipeline execution. Requests mapped to GPUs missing some layers are routed to pipeline-parallel groups that hold complete parameters; KunServe plans which replicas to drop and which groups to form.
- VMM-based local memory remapping. It uses CUDA virtual memory (e.g., cuMemCreate/cuMemMap) so the freed parameter HBM can back KV-cache immediately without touching kernel code.
- Network-aware KV-cache exchange. When a request must continue on a different GPU set, KunServe exchanges KV through the network and prioritizes activation transfer (fine-grained chunks; pause/resume) to avoid stalling the pipeline.
- Bubble-aware batch formulation. After dropping, pipeline bubbles hurt throughput. KunServe introduces lookahead microbatch formation (divide-and-conquer chunking with a cost model) to balance per-stage execution times and keep the pipeline full.