Dynamic Optimizations for LLM Serving: Splitwise & Llumnix

October 10, 2024

Marissa Bhavsar, Shaurya Gunderia, Chris Lin, Raghav Ramesh

Agenda

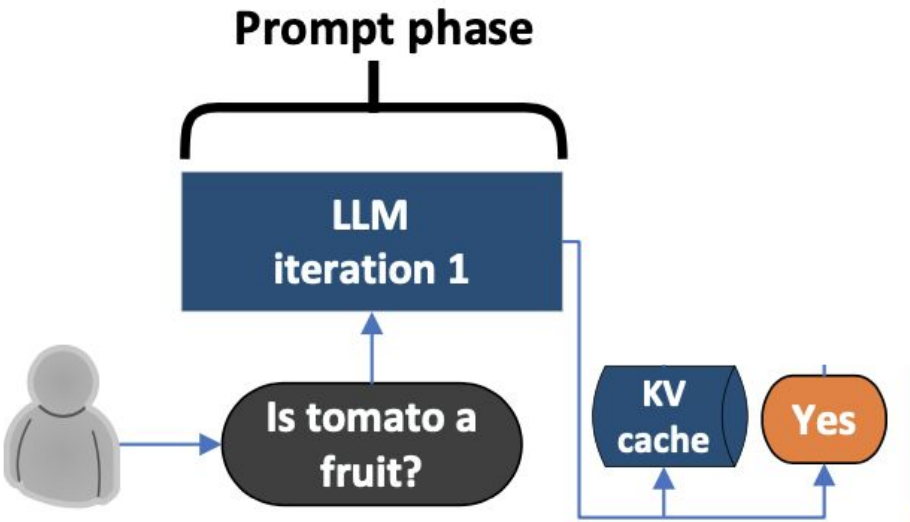
1. Splitwise
    a. Motivation/Background
    b. Inference Insights
    c. Design Overview
    d. Evaluation
2. Llumnix
    a. Motivation/Background
    b. Design + Live Migration
    c. Virtual Usage
    d. Evaluation

Splitwise: Efficient Generative LLM Inference Using Phase Splitting

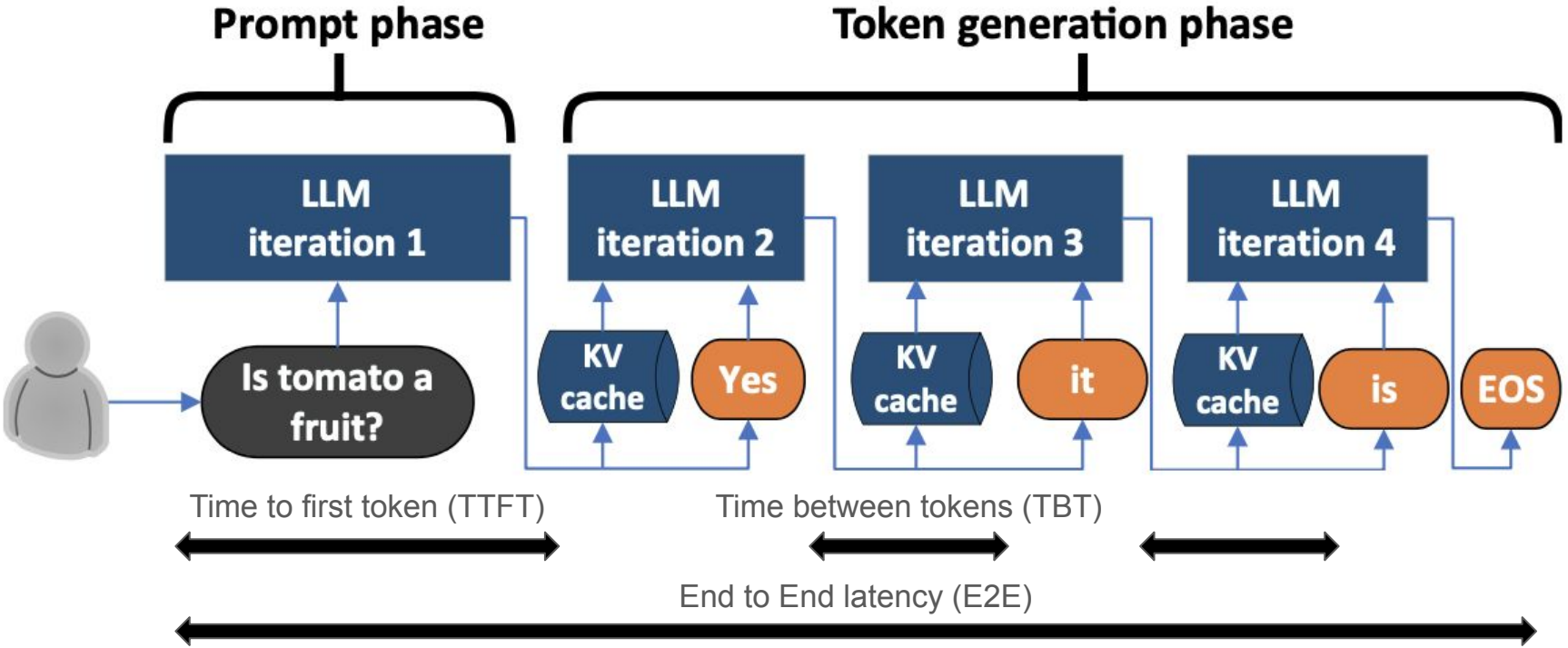
Pratyush Patel[1], Esha Choukse[2], Chaojie Zhang[2],
Aashaka Shah[2], Íñigo Goiri[2], Saeed Maleki[2], Ricardo Bianchini[2]

[1]University of Washington [2]Microsoft

Phases of LLM Inference

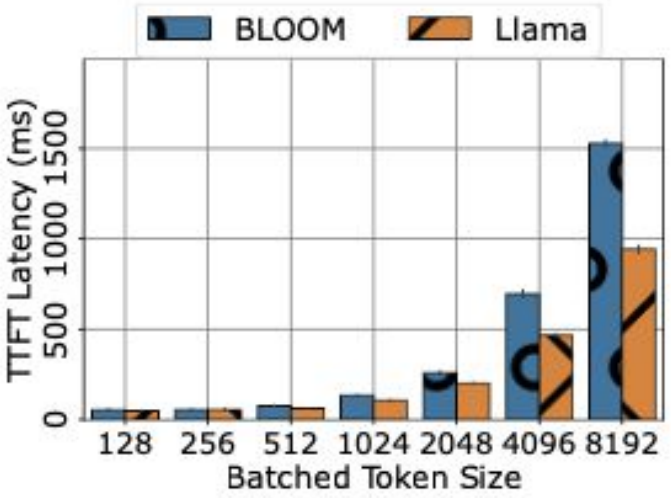


Latency Metrics Used to Measure Performance

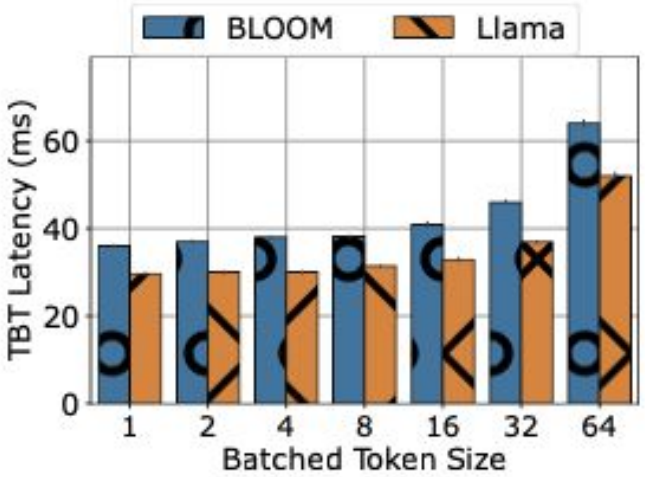


Insights from Current Systems - Varied Latency per Phase

TTFT computed once per request

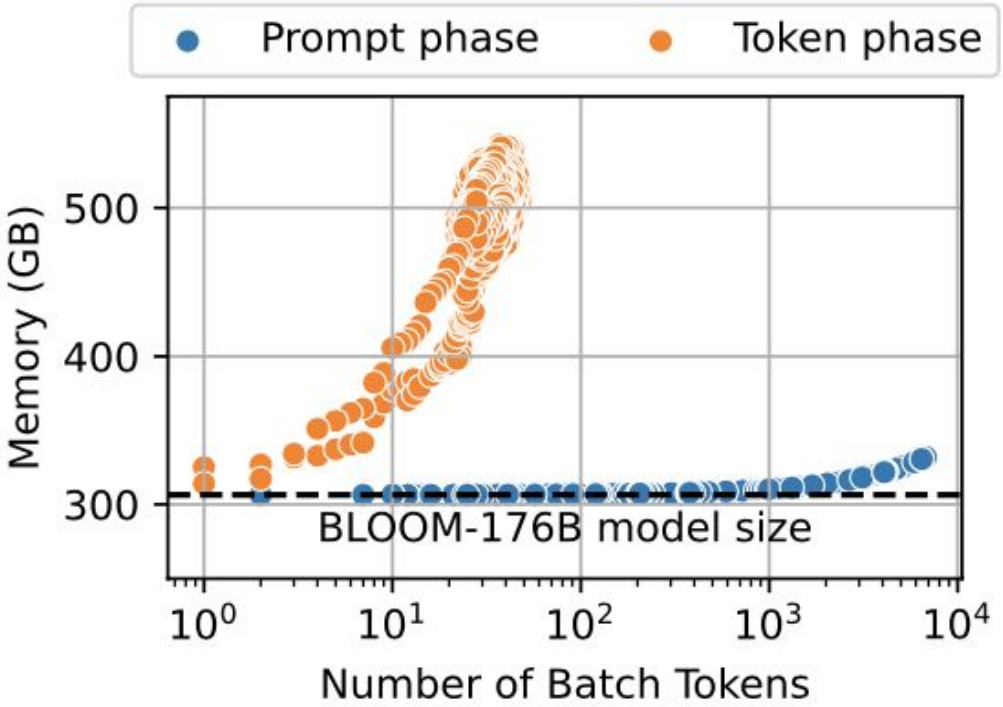


TBT scales with # of generated tokens



Insight: For most requests, the majority of the E2E time is spent in the token generation phase.

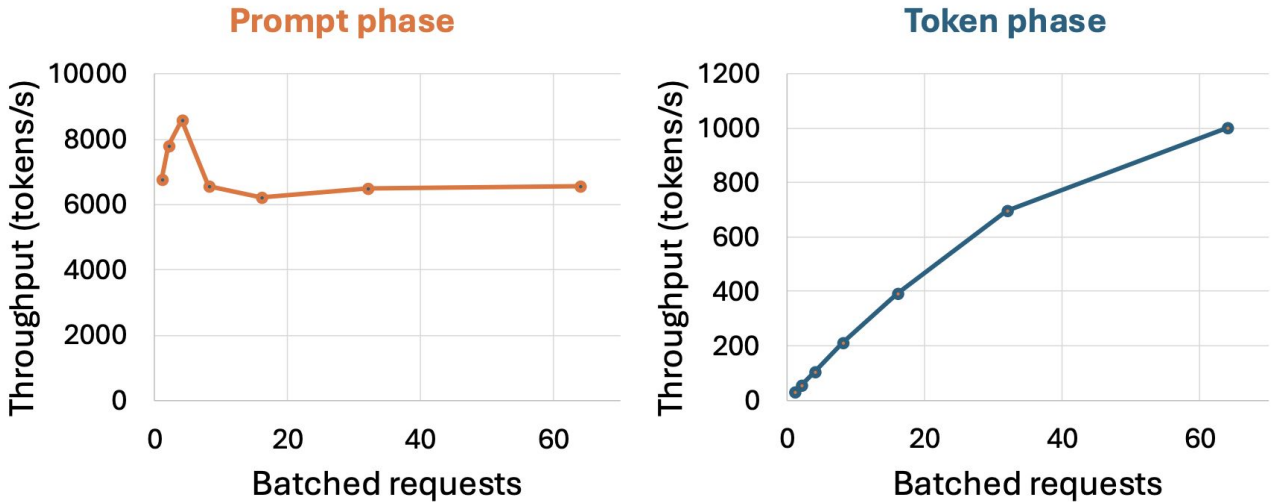
Insights from Current Systems - Memory Requirements



Insight:

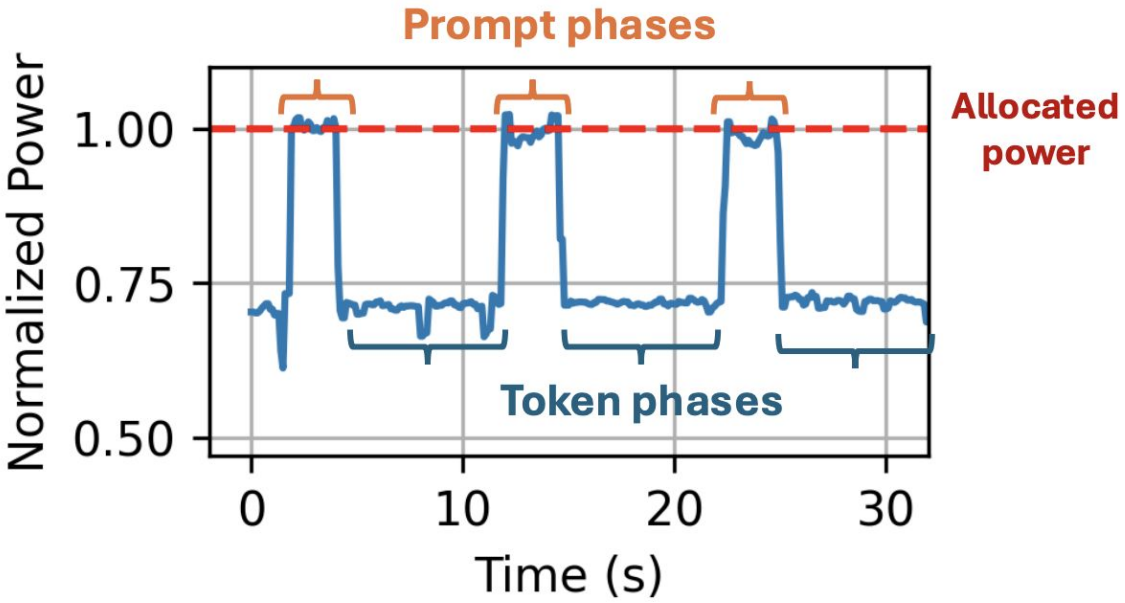
Batching during token generation is limited by memory capacity.

Insights from Current Systems - Throughput Bottlenecks



Insight: Limit the batch size during the prompt phase for better performance; conversely, batching the token generation phase increases throughput without any drawbacks.

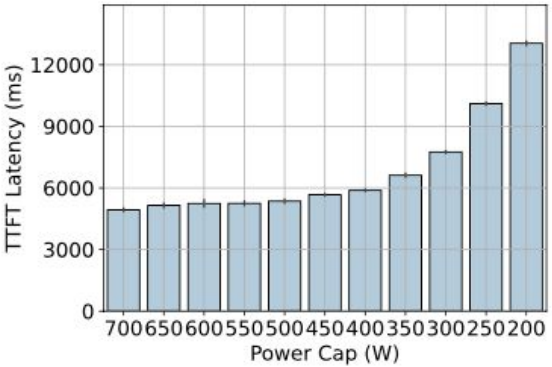
Insights from Current Systems - Varied Power Usage



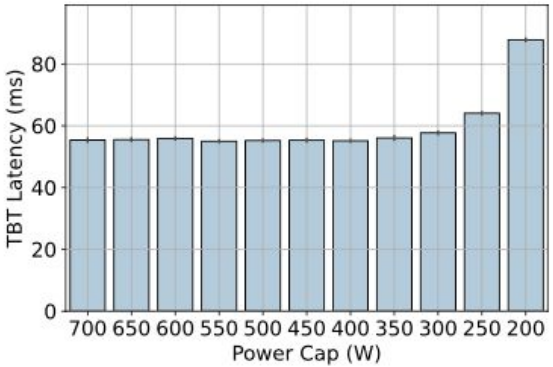
Insight: Prompt phase fully utilized power budget, token phase does not.

*GPU power normalized to thermal design power

Latency vs Power Limitations



(a) Prompt phase.



(b) Token generation phase.

Fig. 9: Impact of power cap on the prompt and token generation latency with the maximum batch size possible.

X-axis: Amount of power used.

Insight: Prompt phase highly sensitive to power capping (2x latency from 700→250 W). TBT remains relatively stable.

Comparison of Inference Phases after Characterization

Prompt Phase	**Token Phase**
Compute and Power Intensive	Memory Intensive (relies on KV Cache)
Highly parallel	Token Generation is Sequential
Need High FLOPs	Low Compute Utilization
Limited batching benefits	Batching improves throughput

Splitwise Motivation

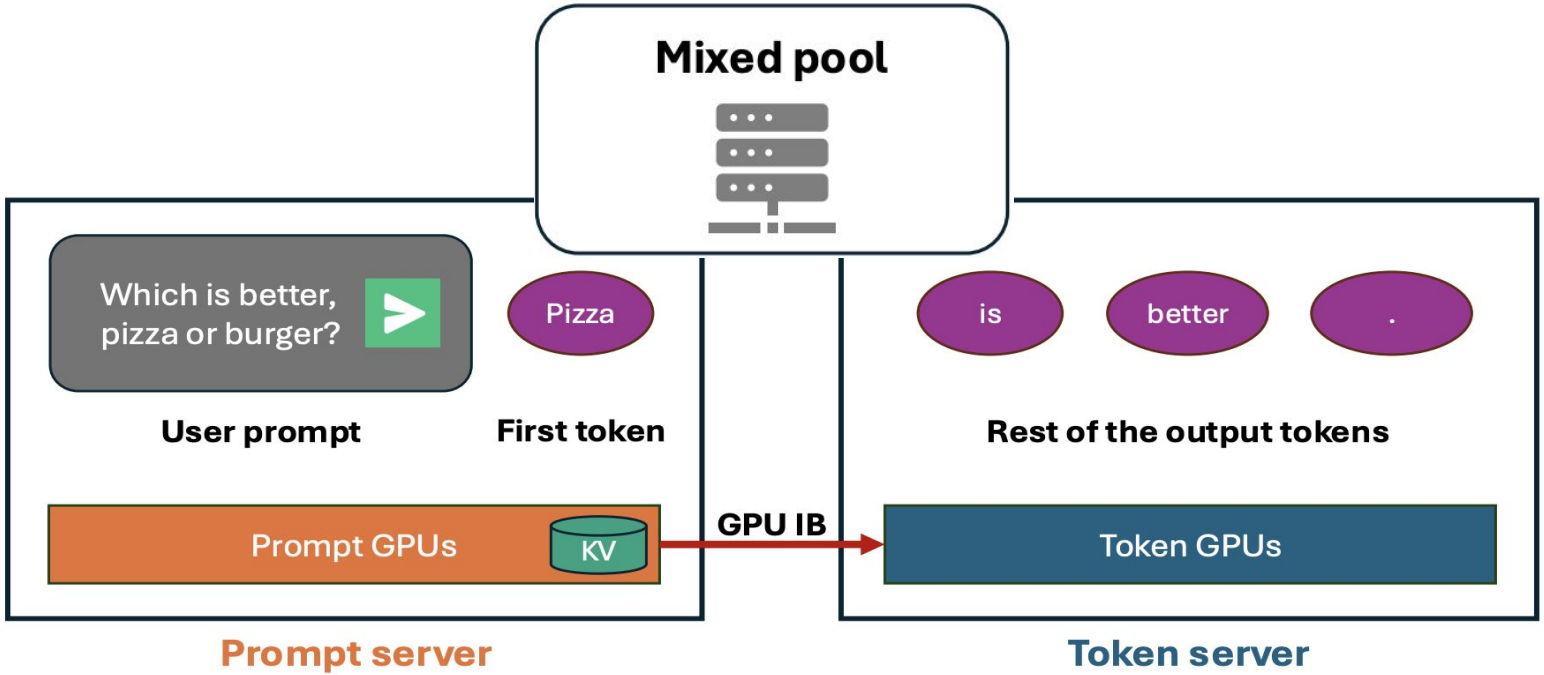
Problem:

- Two phases with distinct differences in latency, memory, throughput, and power
- **Underutilized compute** resources by token generation
- Leads to power inefficiency

Solution:

- Split **two phases** onto **two different** machines
- **Specific** hardware that is well suited for **each task**
- **Throughput & Power** optimized machine configurations by phase

Varied GPU Cluster Design



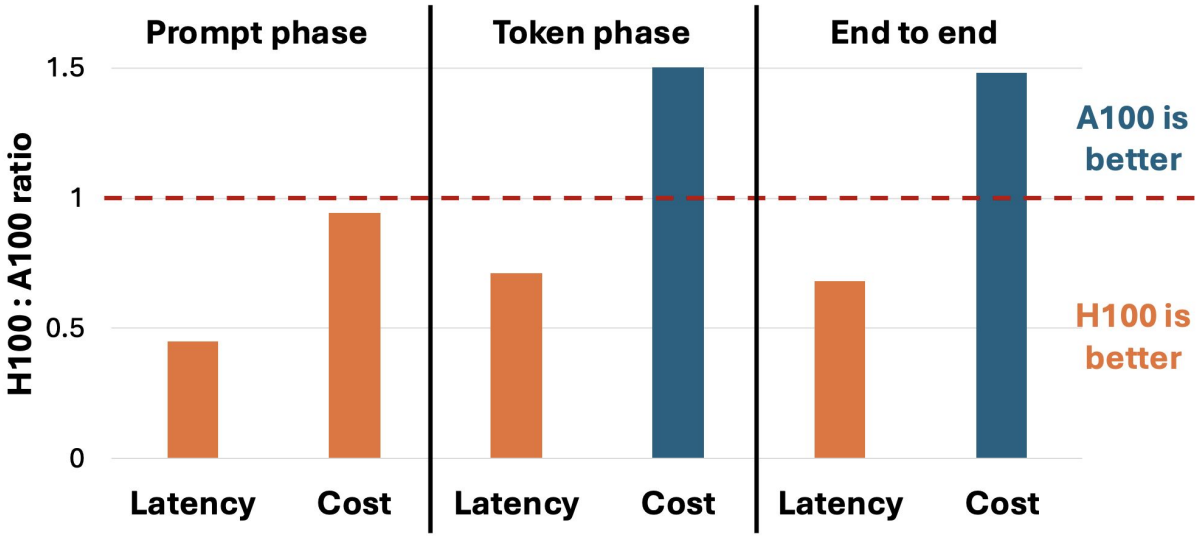
Small batches, maximum power

DGX-H100s

Large batches, power capped

DGX-A100s or DGX-H100s power capped

Different Trade-offs with Different GPUs



Spec	H100 : A100 ratio
Cost	2.15x
Max. power	1.75x
TFLOPs	3.43x

Provisioning With Splitwise

H: DGX-H100

A: DGX-A100

Hcap: Power-capped DGX-H100

Types of Machines – 4 Main Variants

First letter representing the Prompt machine type, and the second letter representing the Token machine type

- Splitwise-HH
- Splitwise-AA

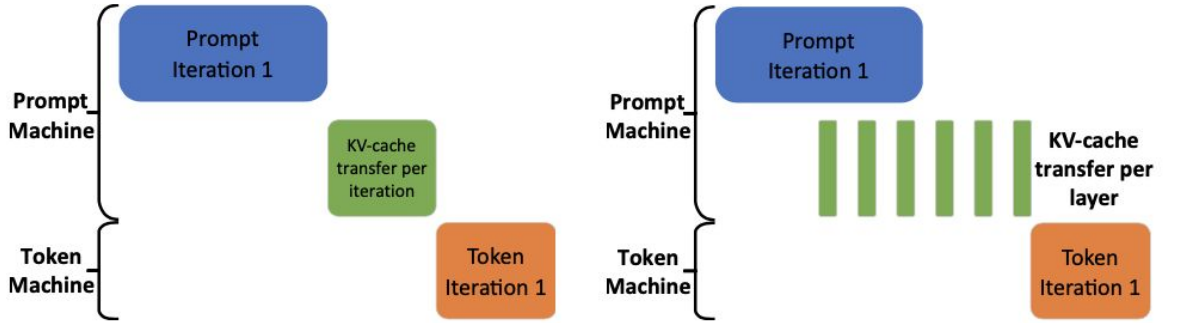
Homogeneous

- Splitwise-HA
- Splitwise-HHcap

Heterogenous

Token generation can be run on less compute-capable hardware

KV Cache Transfer Strategy



(a) Serialized KV-cache transfer. (b) Optimized KV-cache transfer per-layer during prompt phase.

Key Idea: Transfer intermediate KV-cache from prompt to token machines

Computation Overlap: Minimizes transfer delay by transferring during prompt processing

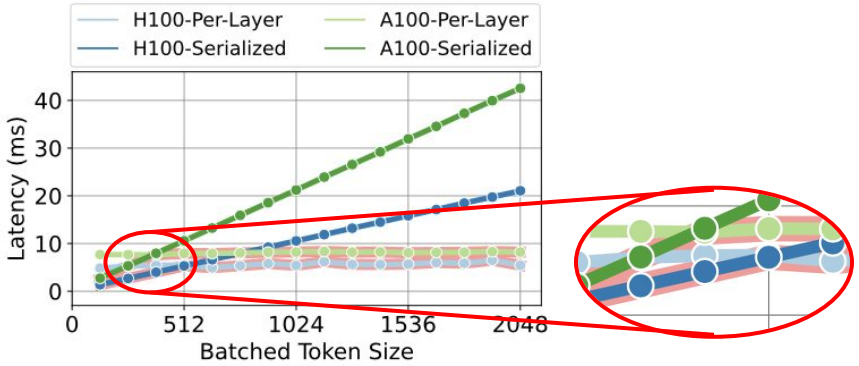
Picks best technique at start of computation

Small Batch Size → Serialized

Large Batch Size → Layer-wise

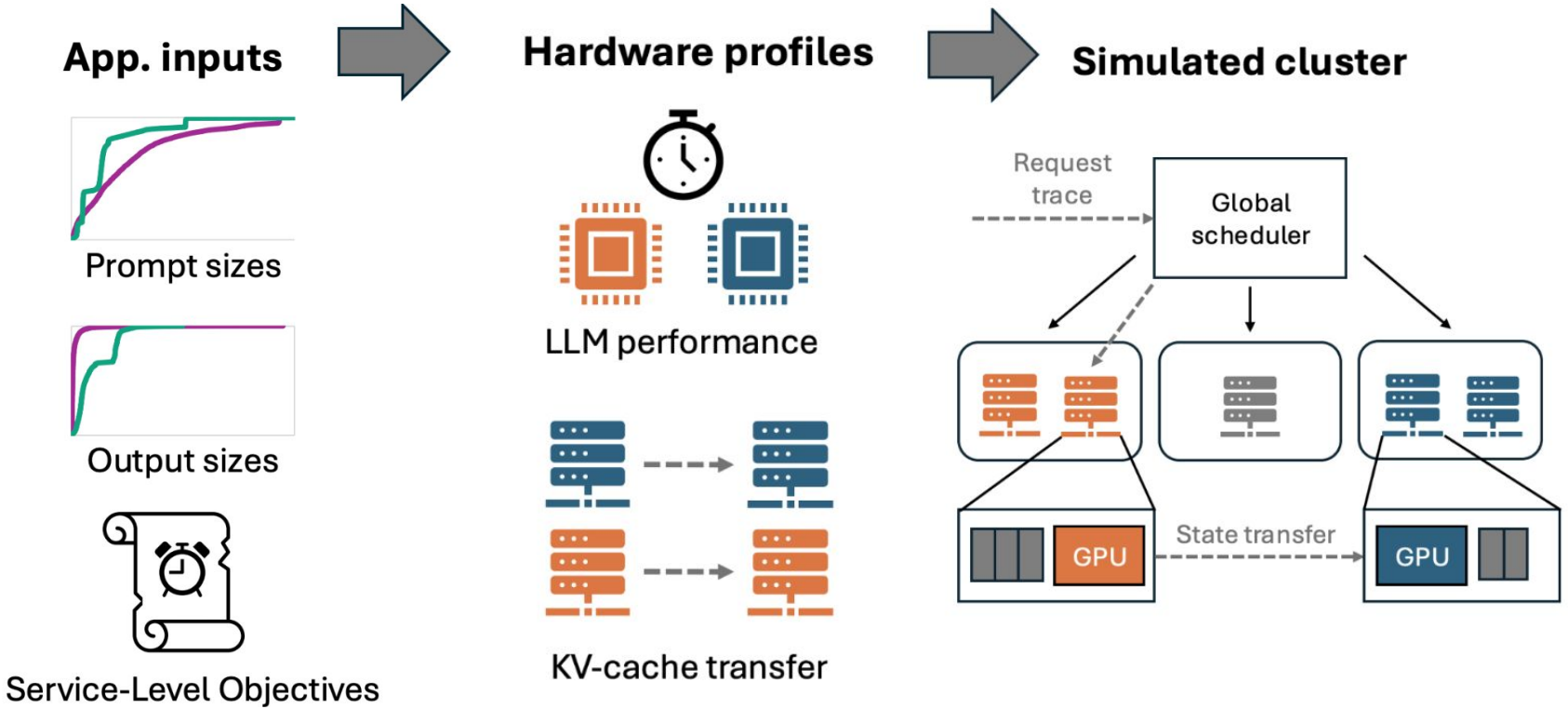
KV Cache Transfer Latency

- < 0.8% overhead for a typical inference request
- Serialized for small prompt sizes (<512) and per-layer for larger prompt sizes



Event-driven cluster simulator

Simulates LLM Inference as a series of events

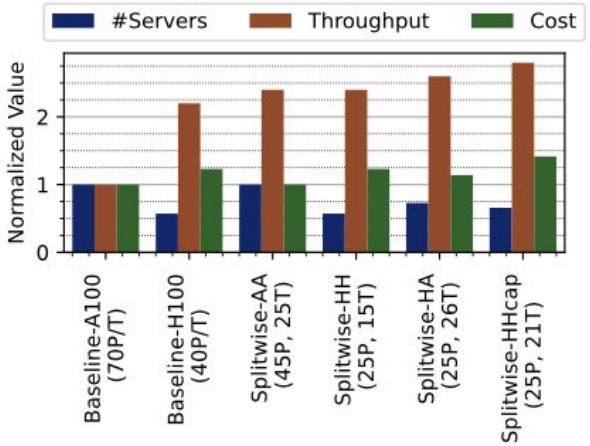


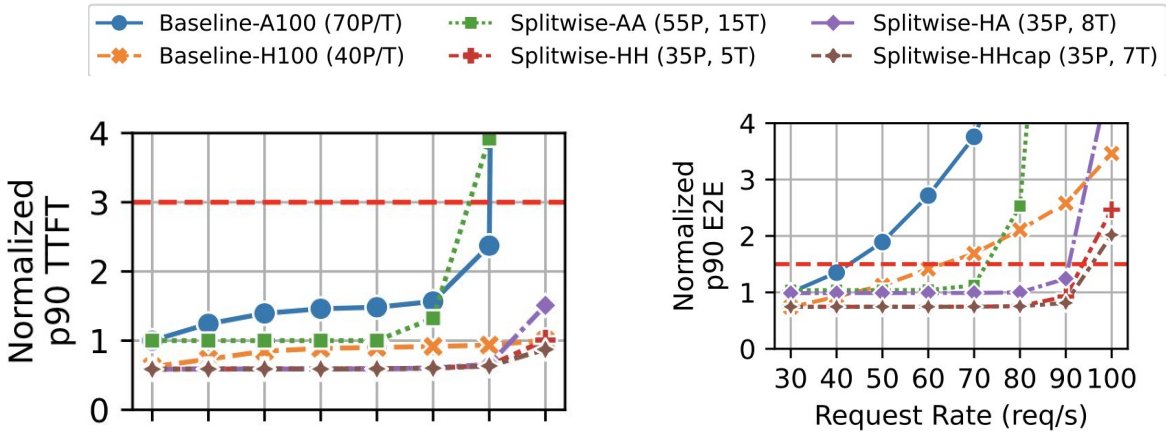
Evaluation

Iso-power Throughput Maximized Clusters

- **Splitwise-HA** bridges gap with low TTFT and E2E at high throughput at lower cost

- **Splitwise-HHcap** excels across all latency and throughput metrics





Drawbacks / Limitations

Reliability & Fault Tolerance

- Failure currently results in Splitwise restarting from scratch
- Checkpoint KV-Cache into an in-memory database

Heterogenous prompt / token machines

- Fragmenting data center with different GPUs may present challenges to CSP
- No infiniband setup for heterogeneous gpu clusters

Future Directions / Considerations

Extensibility to New Models

- All modern transformer-based generative LLMs have these two phases
  - Mixture of Experts

Alternative Compute Hardware

- Methodology applies to any hardware

Conclusion

- Split Inference onto Different Servers for Phase-Specific Resource Management
  - Prompt Phase (computationally expensive, less memory) → DGX-H100s
  - Token Phase (memory intensive, less computation) → DGX-A100s or DGX-H100s power capped

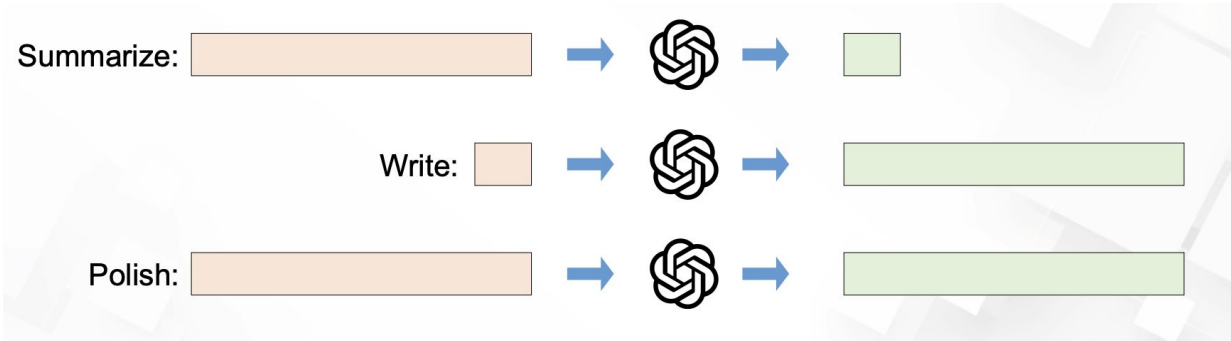
- Splitwise clusters under performance SLOs achieve:
  - 1.76x better throughput with 15% lower power at the same cost
  - 2.35x better throughput with the same cost and power

Llumnix: Dynamic Scheduling for Large Language Model Serving

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao,
Xinyi Zhang, Yong Li, and Wei Lin, *Alibaba Group*

Motivation

- LLM requests are heterogeneous in sequence length and priority
- Autoregressive execution
  - Output sequence lengths are unknown at the time of a request
  - Dynamic GPU memory demands of the KV cache
- Memory Fragmentation (across instances)



Memory Fragmentation -> Preemptions

- Memory Fragmentation: unused memory across multiple instances (inference engines)
- Requests that could have run on this unused memory are instead preempted or blocked

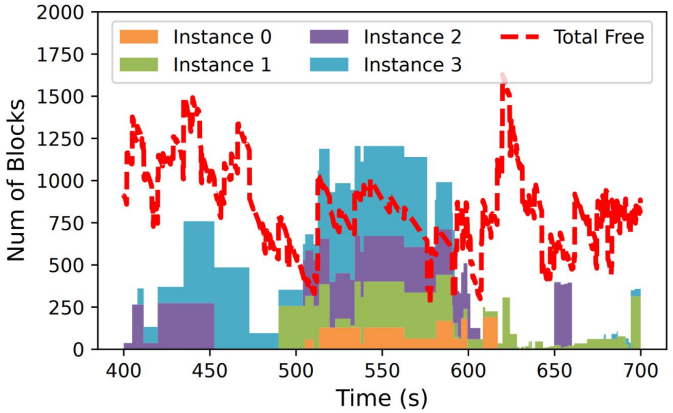


Figure 5: Total free memory vs. demands of the head-of-line queuing requests across four LLaMA-7B instances.

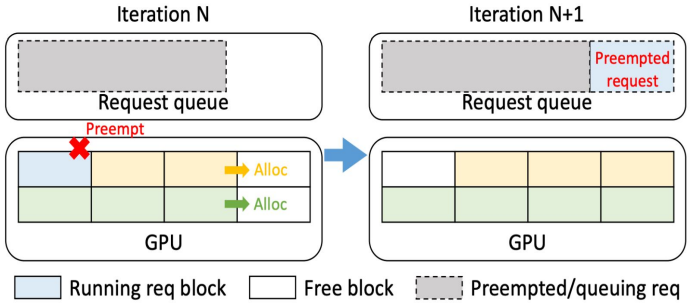
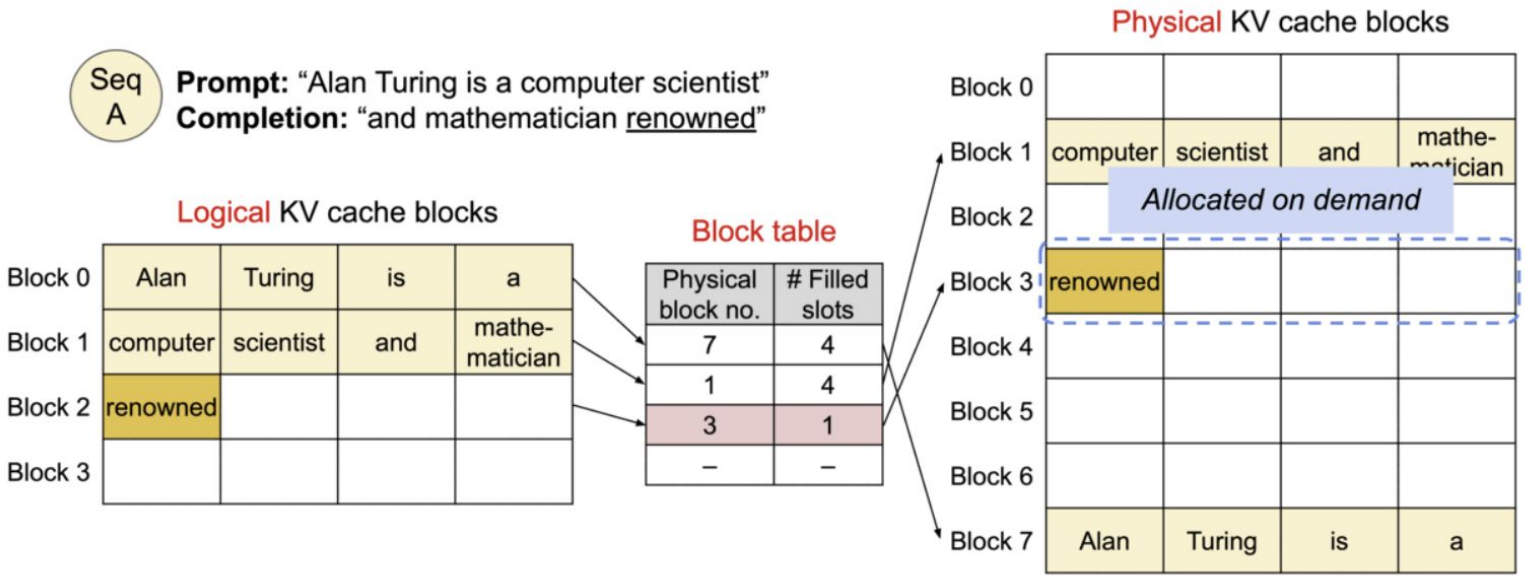


Figure 2: Request queuing and preemption using continuous batching and dynamic memory allocation.

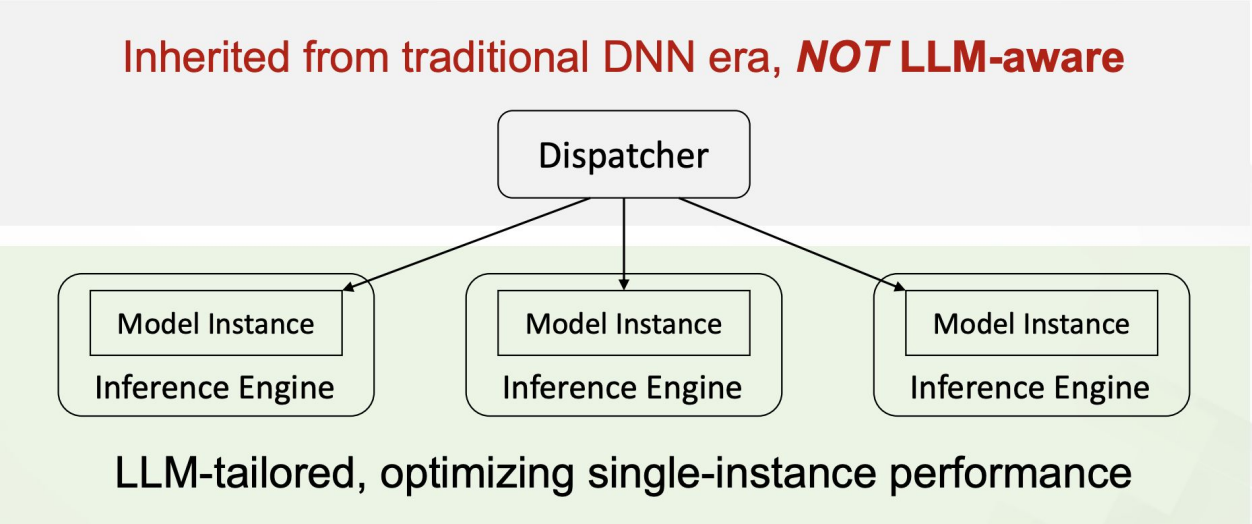
Paged Attention



Example generation process for a request with PagedAttention.

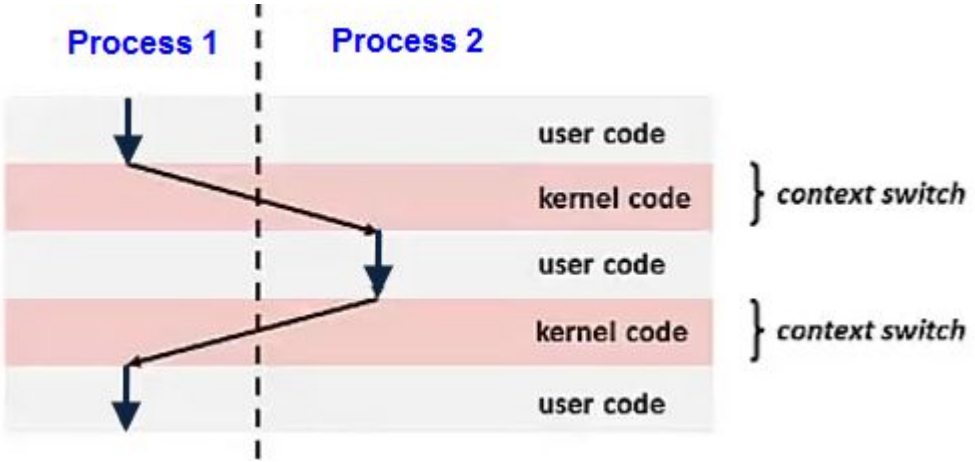
Current LLM serving architecture

- **Static** load balancing of requests (with memory requirements unknown at time of dispatch)



What is Llumnix?

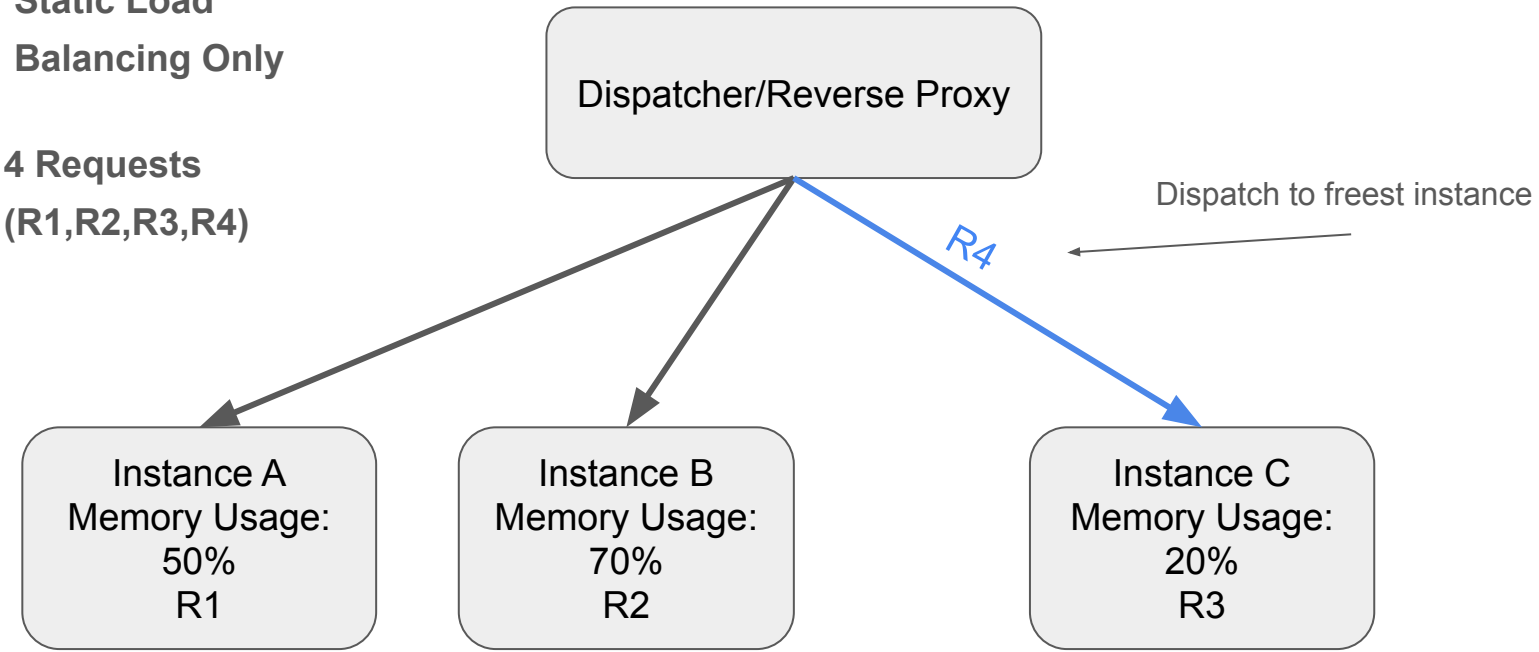
- A dynamic scheduling layer across instances that reschedules requests during computation to more closely align with their dynamic memory demands
  - Resembles context switching in OS whose processes/threads have similar memory patterns
- Built on top of Paged Attention



Without Dynamic Rescheduling

Static Load Balancing Only

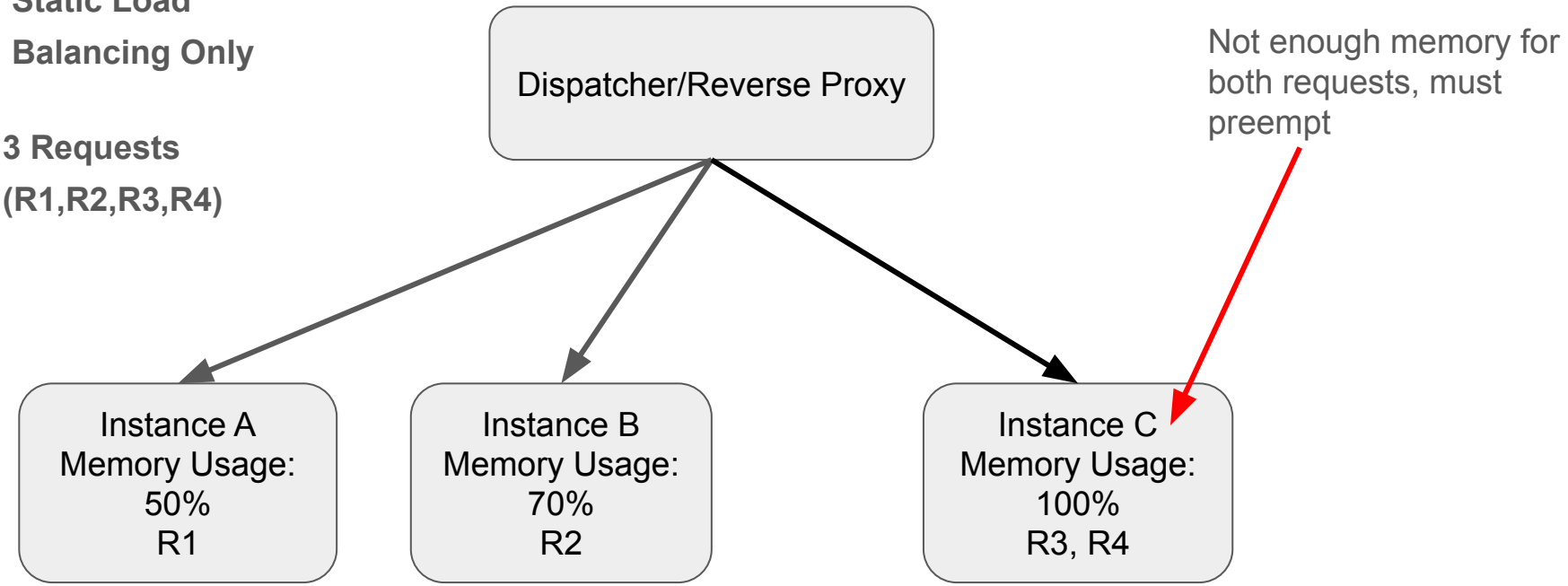
4 Requests (R1,R2,R3,R4)



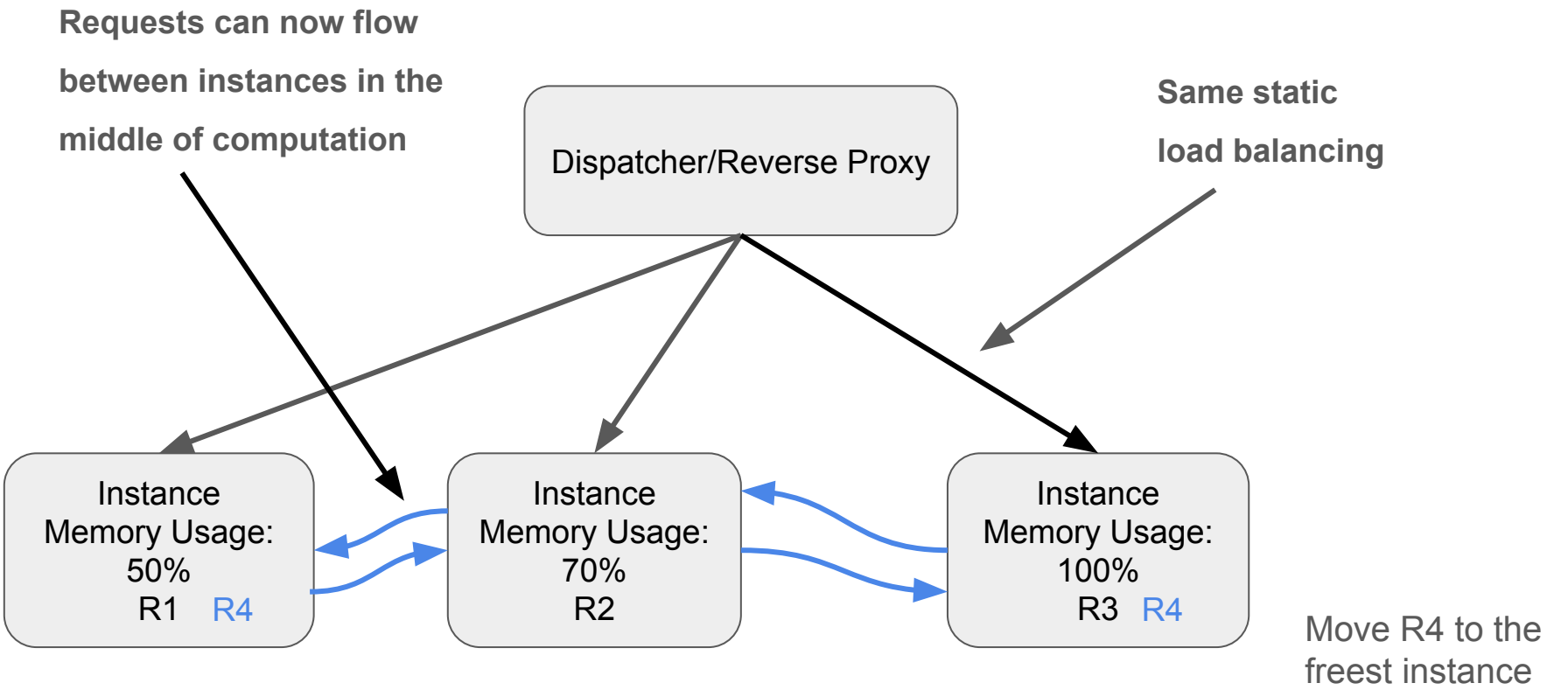
Without Dynamic Rescheduling

Static Load Balancing Only

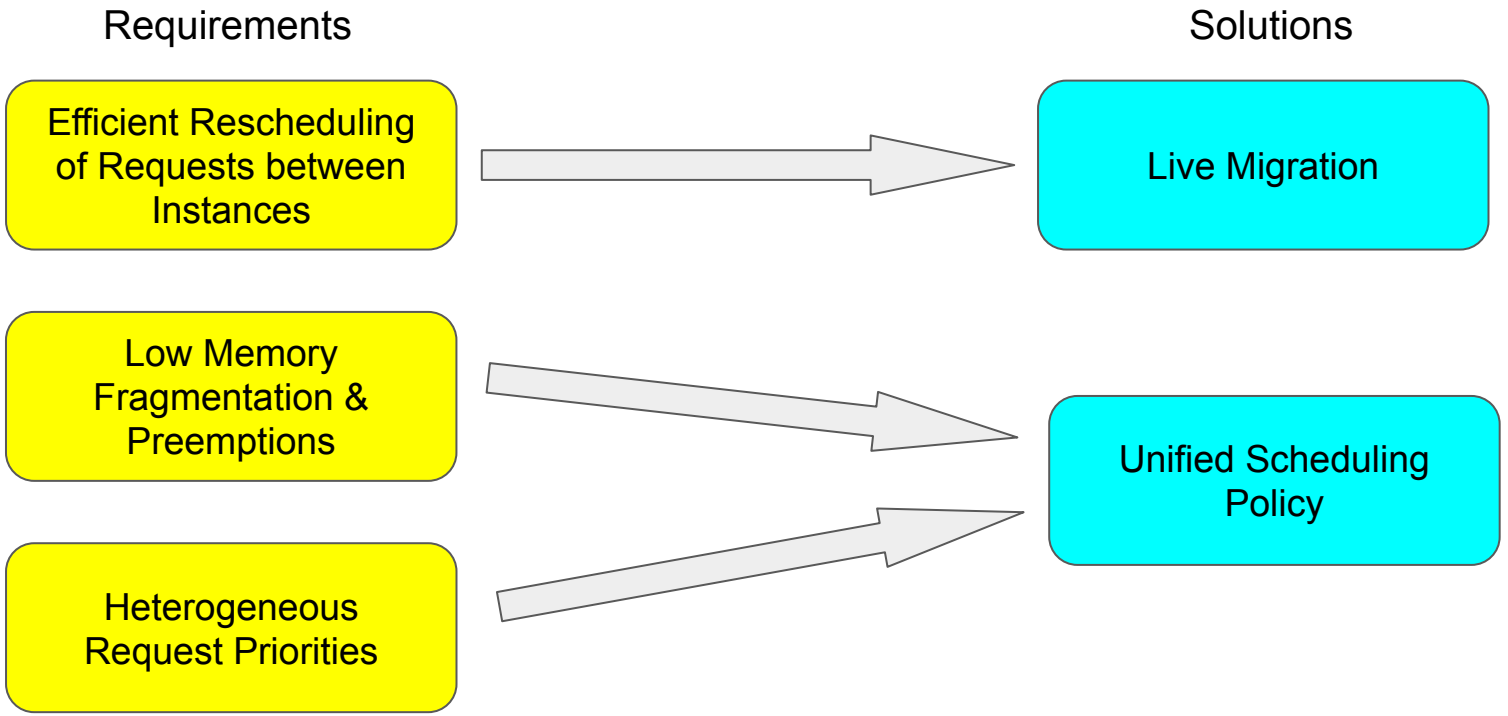
3 Requests (R1,R2,R3,R4)



With Dynamic Rescheduling



How does Llumnix address these motivations?



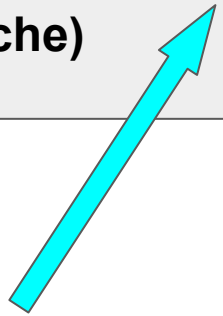
What is in the request context for LLMs?

OS Process

- Stack Pointer
- Registers
- Program Counter
- PID

LLM request

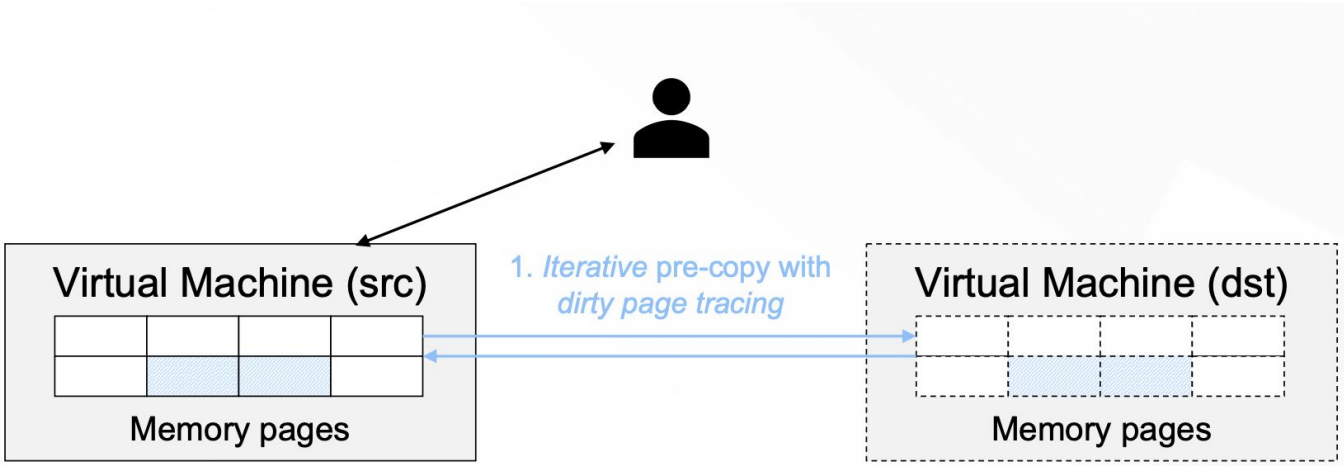
- Input Tokens
- **Model State (KV Cache)**



O(n^2)!!

Live Migration

- Inspiration: VM Live Migration
- Initially copy all pages while src machine is still running

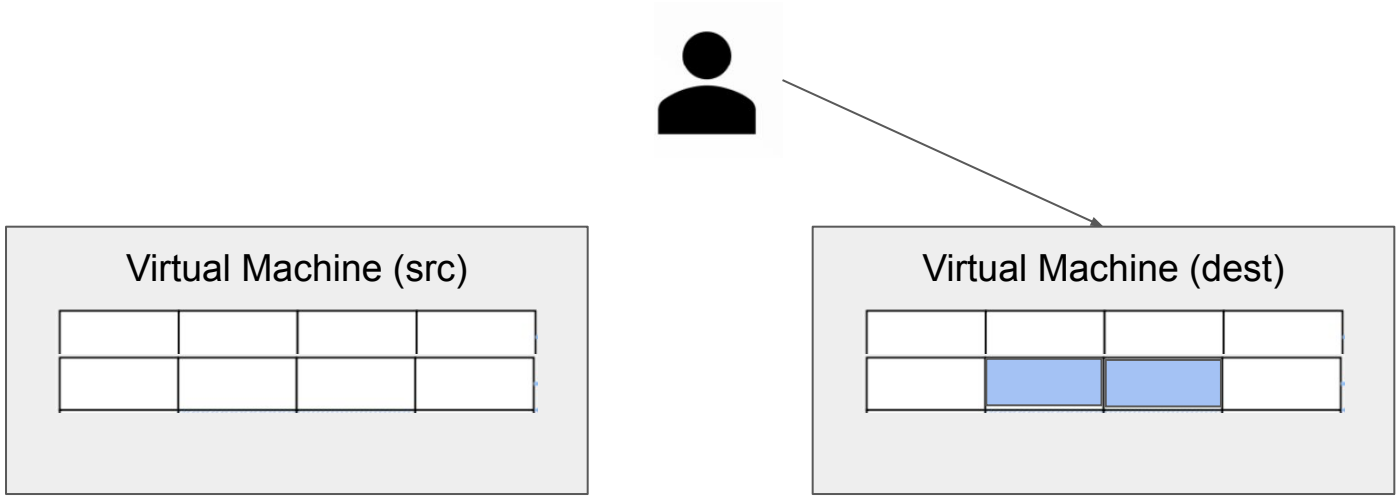


Live Migration

- For every next round, copy any dirty pages in the src
- Once the dirty pages is small, pause execution and copy over remaining dirty pages

Live Migration

- Lastly, suspend execution on src and start executing on dest
- Src memory pages can be freed



Live Migration

- Key characteristic of KV cache: **Append Only**
  - **There are no “Dirty” blocks in the cache**
- Inspired by live migration in VMs
- Pipeline copying of previous KV cache blocks with computation of new blocks
- Downtime is constant to the number of tokens in one iteration

Live Migration

- At every stage, copy KV cache blocks in src to dest
- While copying, calculate the next blocks in the iteration
- At the Nth stage, suspend execution on src and copy remaining block (and rest of request context)
- Resume execution on dest

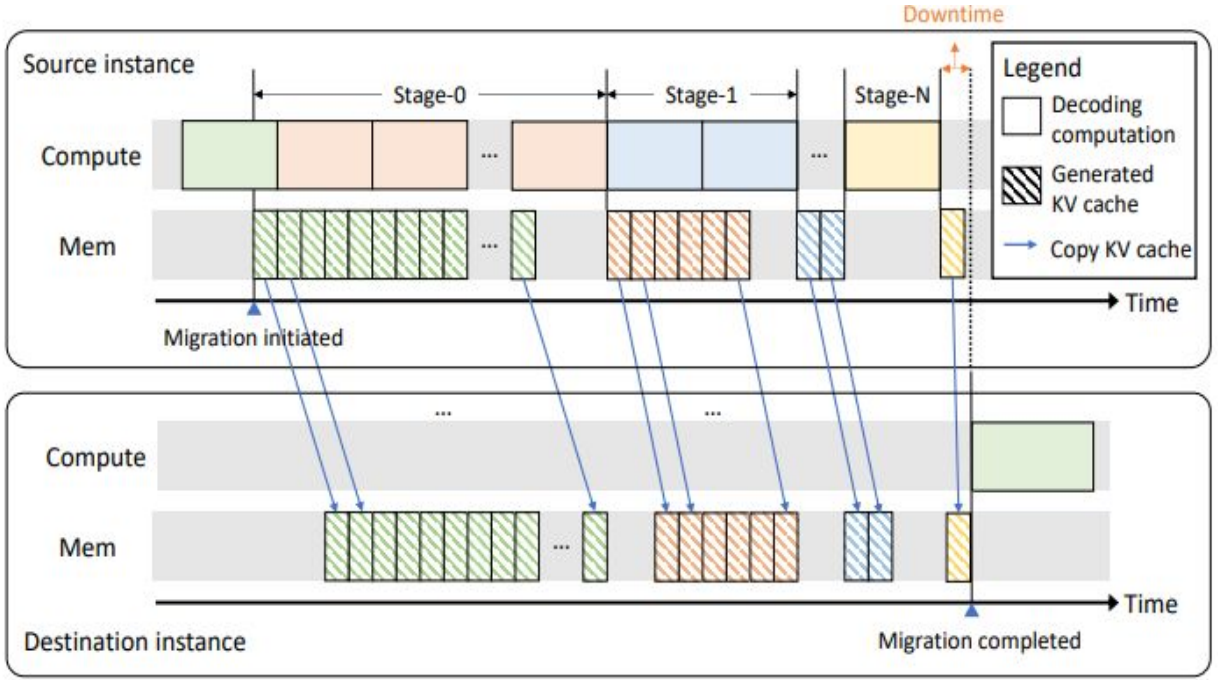


Figure 6: Llumnix adopts multi-stage migration to overlap the computation and KV cache copying for minimal downtime.

Live Migration

- At every iteration, one block of the KV cache is calculated per request
- Iterations (blocks) per stage steadily goes down as copying catches up to compute of new blocks
- The last stage is always one iteration

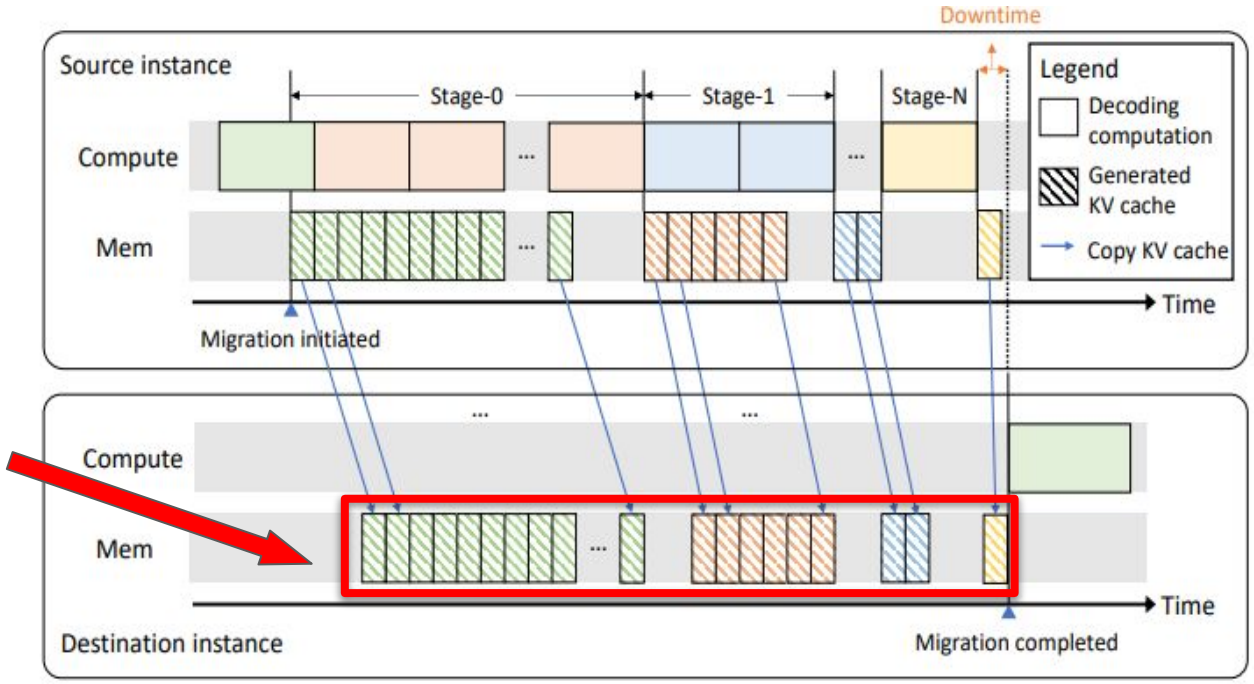


Figure 6: Llumnix adopts multi-stage migration to overlap the computation and KV cache copying for minimal downtime.

Goals of Unified Scheduling Policy

	Goals	Outcomes

1	Reduce prefill and decode latencies	• Minimizing...
		  ○ Queueing delays
		  ○ Preemptions
		  ○ Interference among requests

2	Load adaptivity	• Maintain optimal conditions through instance auto-scaling

3	Single load metric	• Unified metric for:
		  ◦ Load balancing
		  ◦ Prioritization
		  ◦ Queuing delays
		  ◦ Creating free space on instance

Goals of Unified Scheduling Policy

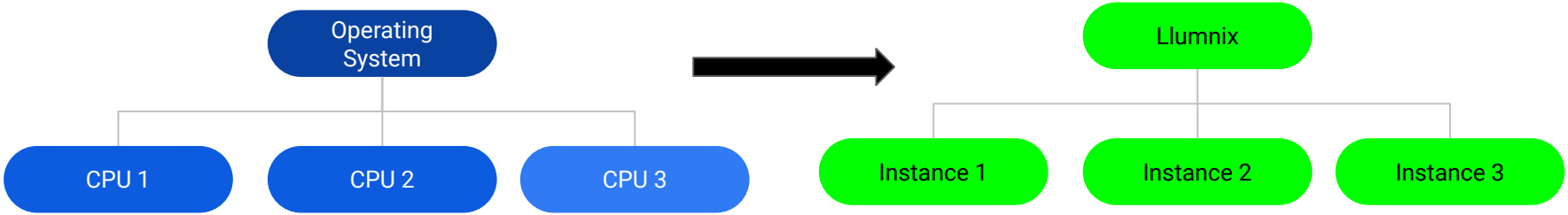
	Problem	Consequences
1	High prefill and decode latencies	• Increased queuing delays, preemptions, and interference among requests • Slower response for users
2	Inability to adapt to changing load	• Underutilized/overloaded instances • Higher costs • Degraded performance
3	Lack of request prioritization	• Processing delays for high-priority requests • No distinction between different priority requests

Solution: Virtual Usage

Virtual Usage: Abstraction layer built upon physical usage acting as unified metric for the amount of resources necessary for requests/instances

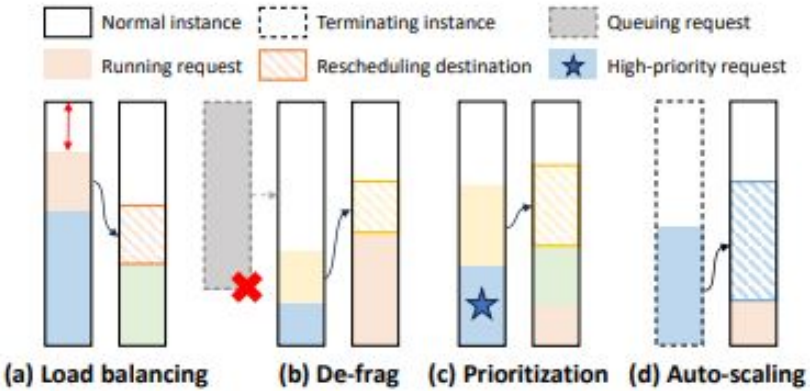
Problems → Solution

- High prefill + decode latencies → proactive + dynamic resource allocation to reduce queuing delays and preemption
- Lack of load adaptability → dynamic view of resource usage allows for better load balancing
- Lack of request prioritization → higher-priority requests ‘appear’ to need more resources
- Similar to OS task distribution to CPUs → estimating resource consumption for processes and threads

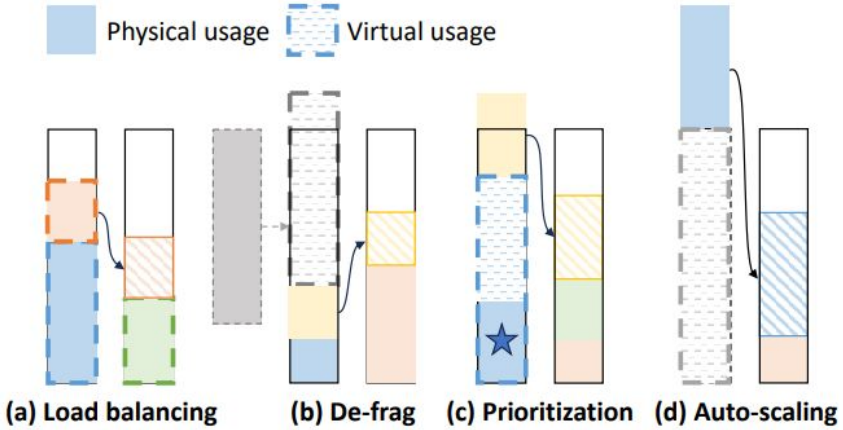


Scheduling Scenarios of Virtual Usage

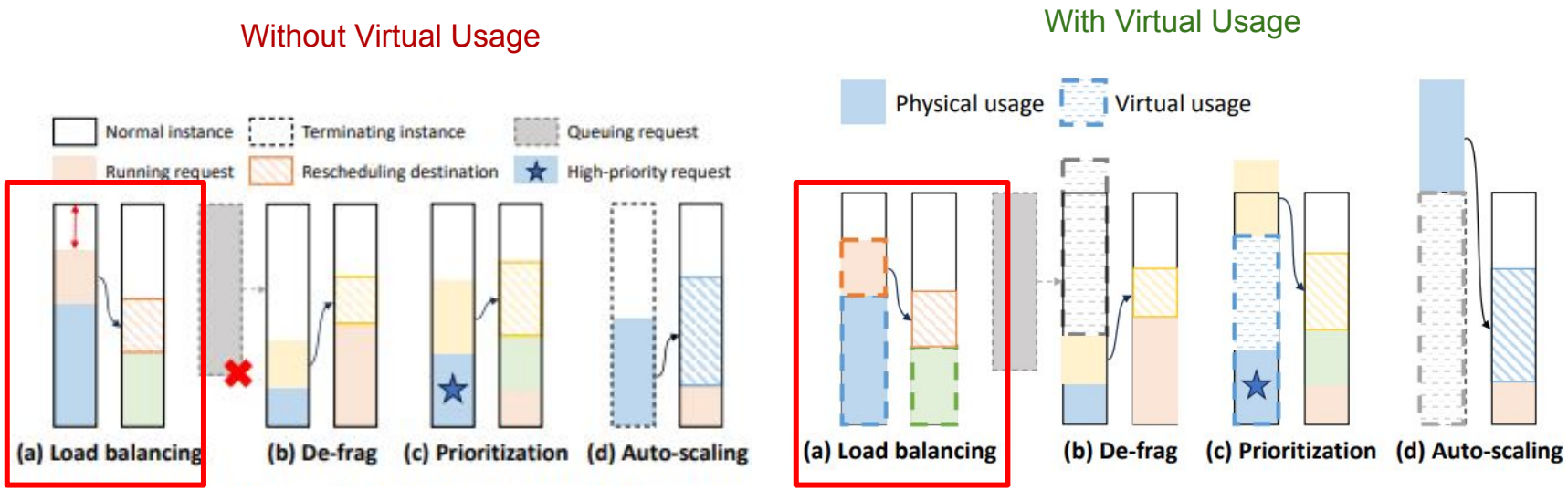
Without Virtual Usage



With Virtual Usage



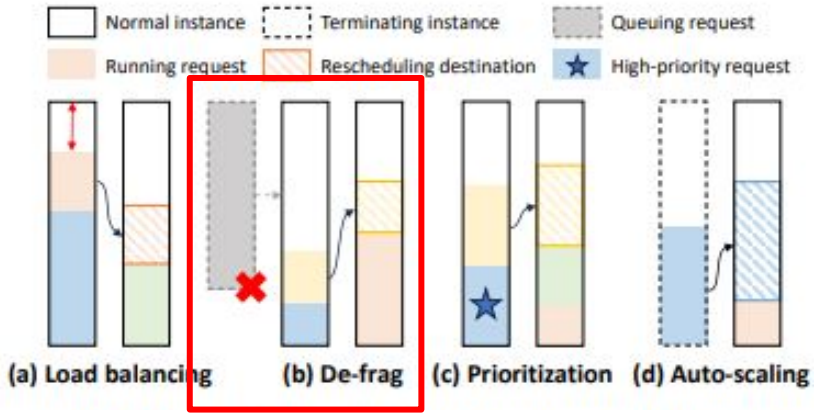
Scheduling Scenarios of Virtual Usage



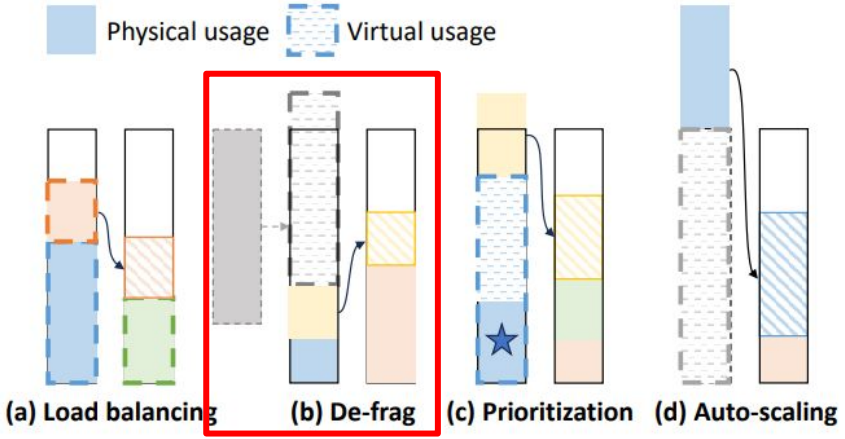
Load Balancing: virtual usage = physical usage, **Advantage: Requests dispatched to instances with sufficient virtual memory**

Scheduling Scenarios of Virtual Usage

Without Virtual Usage



With Virtual Usage

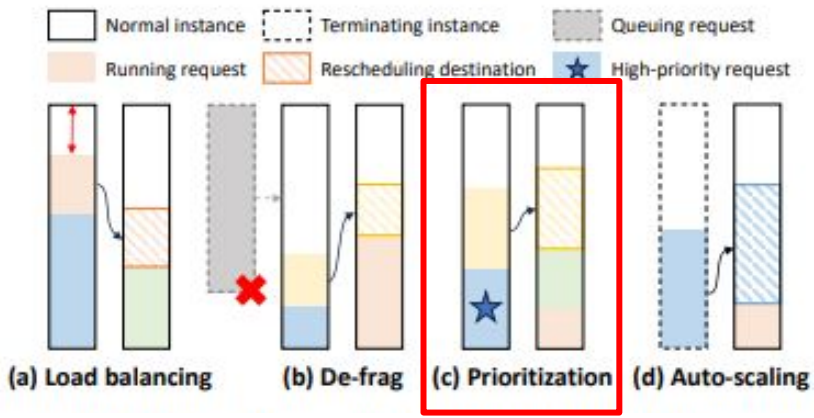


Load Balancing: virtual usage = physical usage, **Advantage: Requests dispatched to instances with sufficient virtual memory**

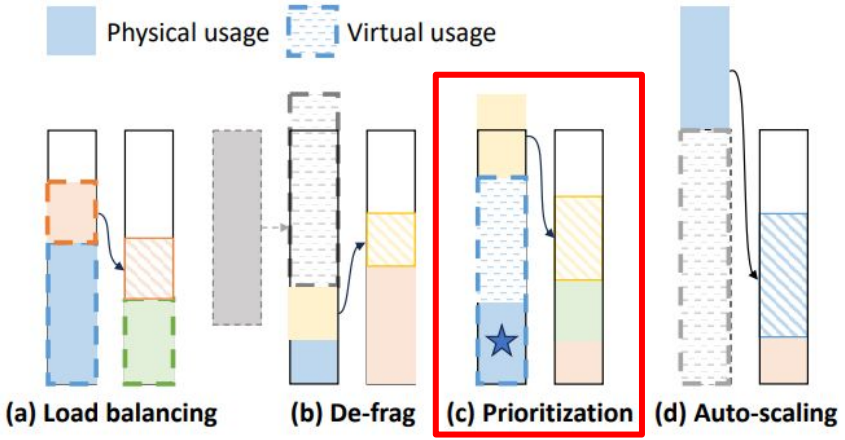
Defragmentation: virtual usage of new request exceeds available resources, **Advantage: Reduce queuing delays**

Scheduling Scenarios of Virtual Usage

Without Virtual Usage



With Virtual Usage

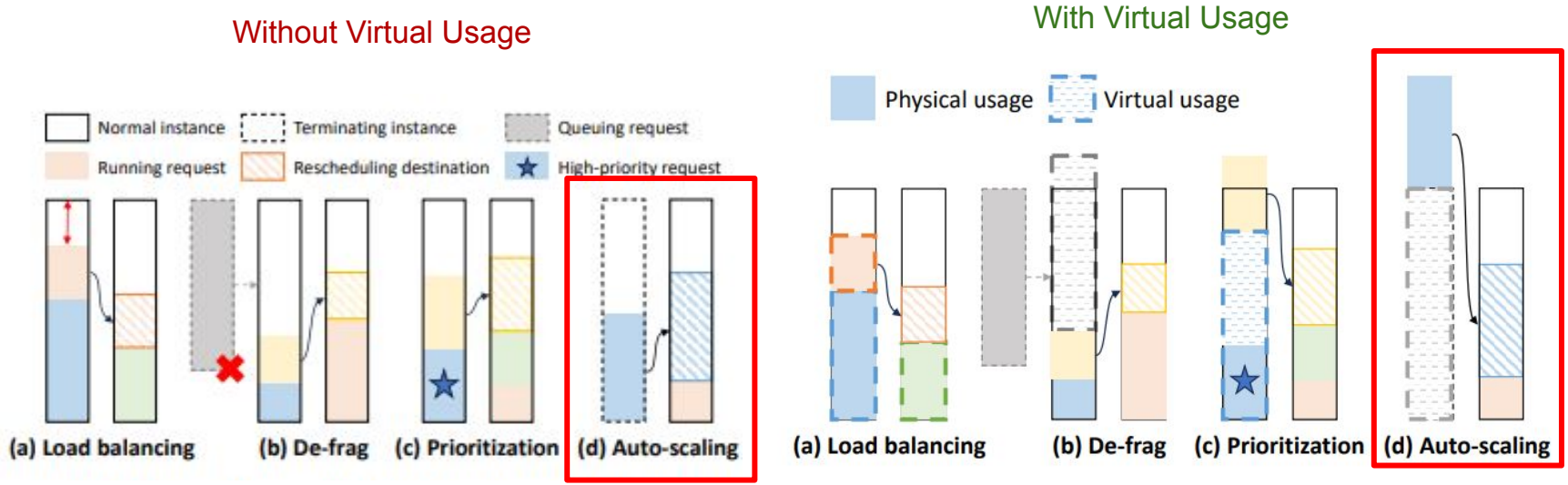


Load Balancing: virtual usage = physical usage, **Advantage: Requests dispatched to instances with sufficient virtual memory**

Defragmentation: virtual usage of new request exceeds available resources, **Advantage: Reduce queuing delays**

Prioritization: High priority requests get extra ‘headroom’, **Advantage: Less resource interference with high priority requests**

Scheduling Scenarios of Virtual Usage



Load Balancing: virtual usage = physical usage, **Advantage: Requests dispatched to instances with sufficient virtual memory**

Defragmentation: virtual usage of new request exceeds available resources, **Advantage: Reduce queuing delays**

Prioritization: High priority requests get extra ‘headroom’, **Advantage: Less resource interference with high priority requests**

Auto-scaling: an instance can be overloaded causing it to get drained, **Advantage: Proactive scaling before getting overloaded**

Dispatching using ‘Freeness’

“Freeness”: How much computational resources are available to process new requests on a given instance

$$F = (M - \sum V)/B$$

F: Freeness

M: Total memory

V: Virtual usage of each request

B: Batch size

Dispatching using ‘Freeness’

“Freeness”: How much computational resources are available to process a new request on a given instance

$$F = (M - \sum V)/B$$

F: Freeness

M: Total memory

V: Virtual usage of each request

B: Batch size

Why are we dividing total memory available by B??

In order to get the amount of resources allocated for 1 request in that batch

High freeness → more available resources for processing requests

Low freeness → less available resources to process a request

Experimental Setup

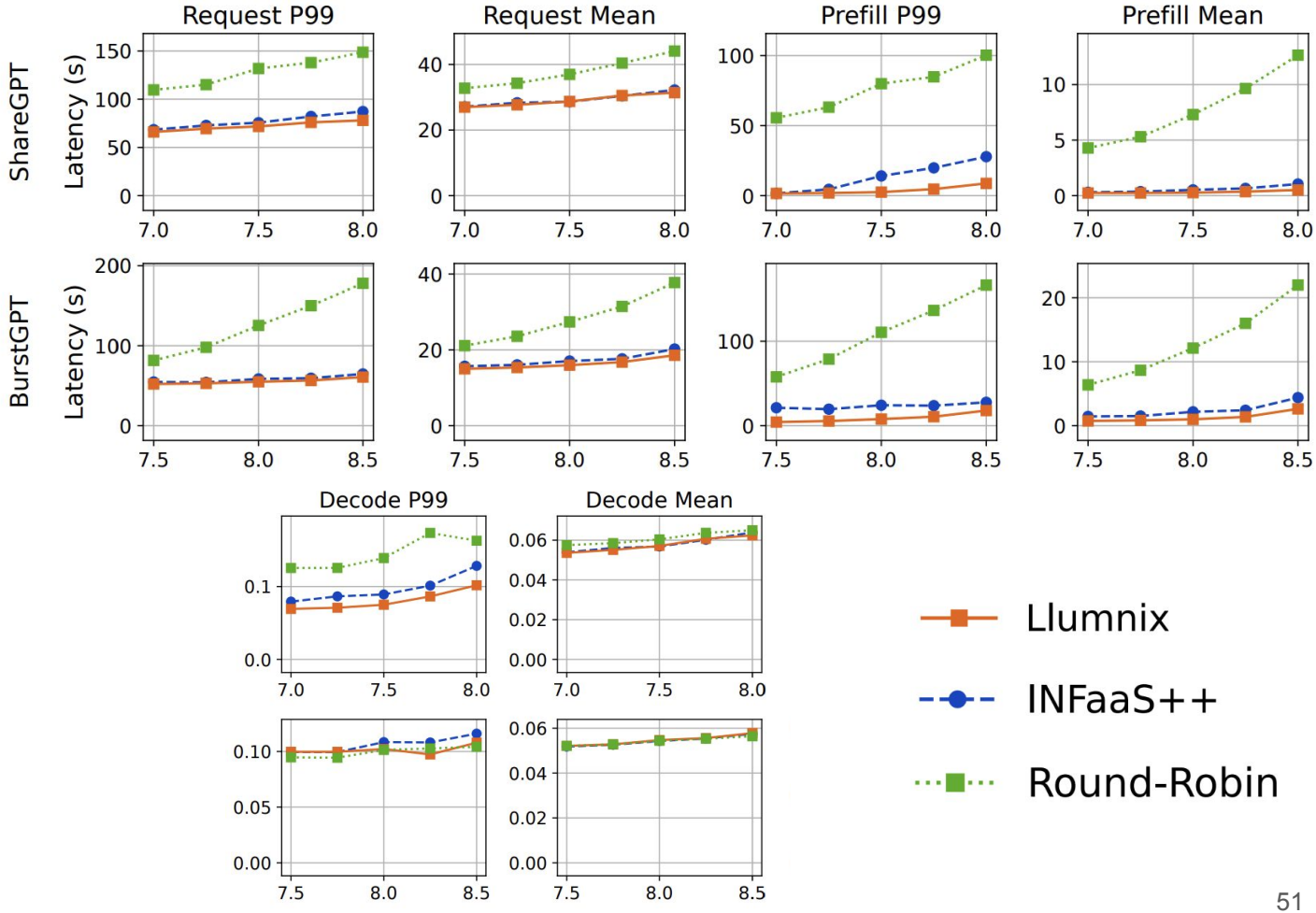
- Models: LLaMA-7B and LLaMA-30B
- Datasets: ShareGPT, BurstGPT, generated datasets (S, M, L)
- Baselines:
  - Round-robin dispatching
  - INFaaS++

Serving Performance (Real Datasets)

End-to-end: 2x/2.9x for mean and P99 over baselines

Prefill: 2.2x/5.5x for mean and P99 over INFaaS++

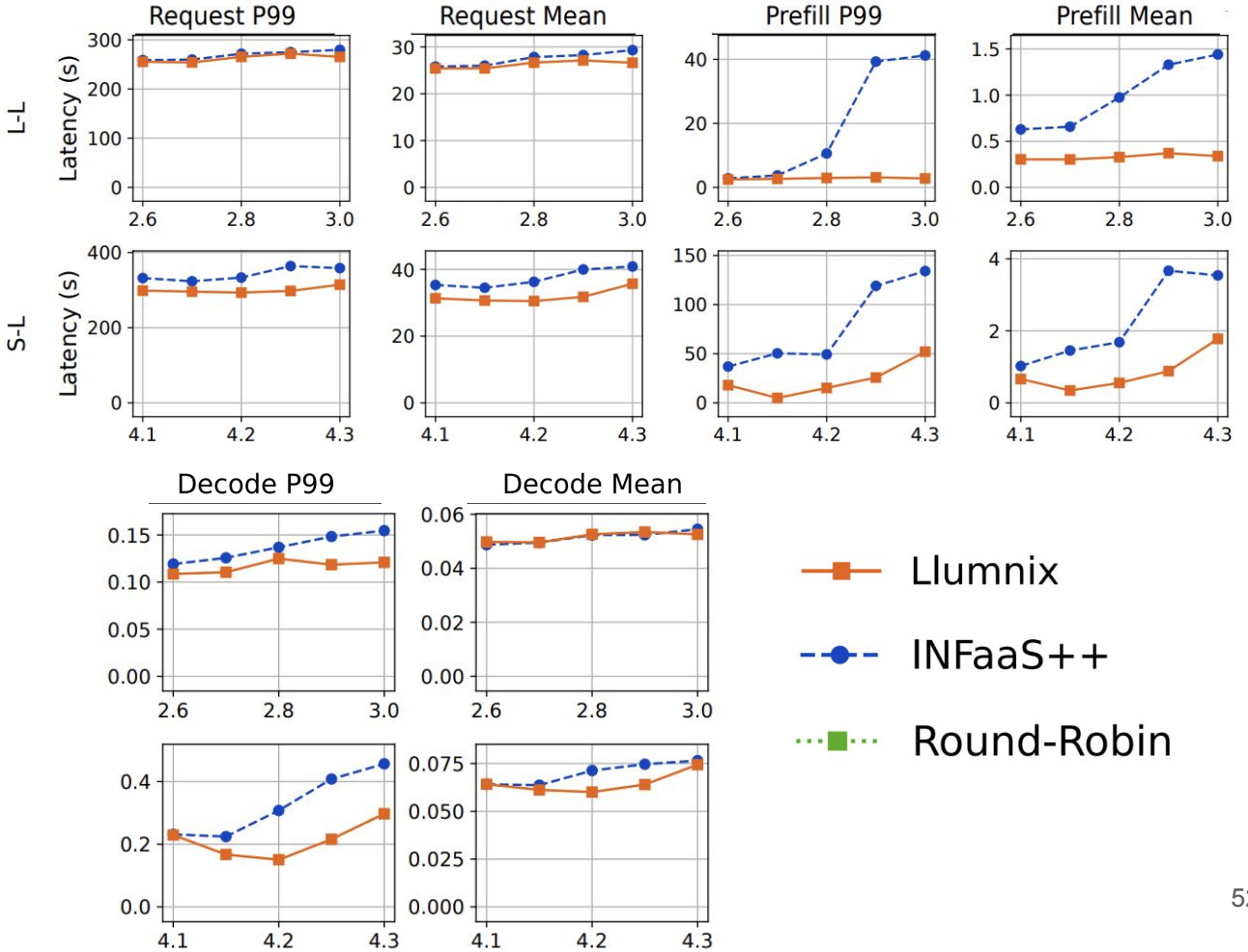
Decode: 1.33x for P99 over INFaaS++



Serving Performance (Generated Datasets)

End-to-end: 1.5x/1.6x for mean and P99 over INFaaS++

Prefill: 7.7x/14.8x for mean and P99 over INFaaS++



Future Work

- Support for multiple model types
  - Currently supports LLMs
  - Extension examples: Fine-tuned models, non-Transformer models
- Enhanced migration policies
  - Handling performance isolation, memory efficiency, load balancing
- Global scheduling vs local scheduling
  - Possibly combining or coordinating to two levels of scheduling

Llumnix Design

- Provides a dynamic rescheduling layer between instances
  - Near-zero downtime that is constant to sequence lengths of the requests
- Adds priorities to requests; important requests will be preempted less
- Unified scheduling policy to handle fragmentation, priorities, preemption, load balancing called “virtual usage”
- Transfer request contexts (KV cache) with low downtime using live migration

Live Migration Handshake Process

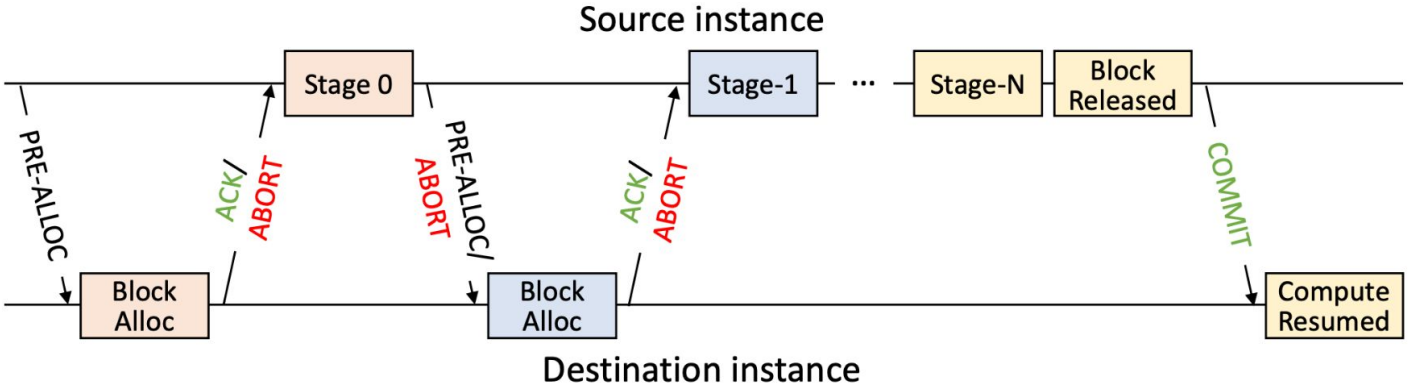


Figure 7: Handshake during migration.

Distributed Scheduling Architecture

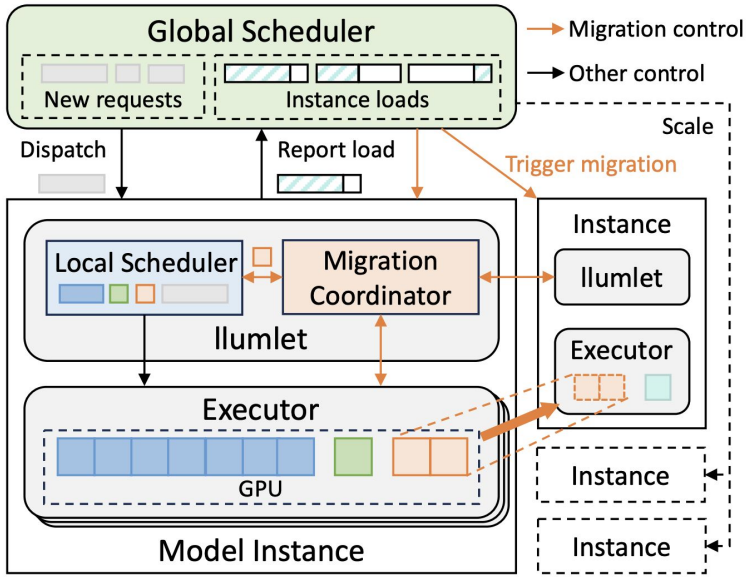


Figure 8: Llumnix architecture.

- Cluster level global scheduler
  - Operates on the level of instances, load balancing according to instance memory usage
  - Dispatches new requests, triggers migrations, and controls instance auto-scaling
- Instance level scheduler (llumlets)
  - Each instance runs several requests
  - Reports to the global scheduler the aggregate memory usage of all its requests

Unified Scheduling Policy

Goals:

1. Reduce prefill and decode latencies by minimizing queuing delays, preemptions, and interference among requests
2. Load adaptivity, aiming to maintain optimal conditions through instance auto-scaling
3. To integrate various (sometimes) conflicting goals into a single load metric for instances
    - Load balancing vs Prioritization vs Queuing delays vs Creating free space on instance

Physical Usage vs Virtual Usage

Feature	Physical Usage	Virtual Usage
Definition	Actual resources currently in use	Abstract representation of resource demand
Measurement	Directly from systems metrics	Calculated based on rules/policies
Nature	Static	Dynamic
Implication	Reflects real-time resource consumption	Affects scheduling and resource management decisions

Virtual Usage + Freeness Algorithm

Uses simple rule where **virtual usage = physical usage**

Algorithm 1: Virtual Usage and Freeness Calculation

```
1  Function CalcVirtualUsage(req, instance):
2      if req.isQueuing then
3          if req.isHeadOfLine then
4              return req.demand
5          return 0
6      if req.isFake then
7          return ∞
8      return req.physicalUsage+GetHeadroom(req.priority, instance)
9  Function GetHeadroom(p, instance):
10     return headroomForPriority[p]/instance.numRequests[p]
11 Function CalcFreeness(instance):
12     if instance.isTerminating then
13         AddFakeReq (instance.requests)
14     totalVirtualUsages = 0
15     for req in instance.requests do
16         totalVirtualUsages+ =CalcVirtualUsage(req, instance)
17     freeness = (instance.M − totalVirtualUsage)/instance.B
18     return freeness
```

Virtual Usage + Freeness Algorithm

High priority requests get extra **headroom** added to virtual usage

Algorithm 1: Virtual Usage and Freeness Calculation

```
1  Function CalcVirtualUsage(req, instance):
2      if req.isQueuing then
3          if req.isHeadOfLine then
4              return req.demand
5          return 0
6      if req.isFake then
7          return ∞
8      return req.physicalUsage+GetHeadroom(req.priority, instance)
9  Function GetHeadroom(p, instance):
10     return headroomForPriority[p]/instance.numRequests[p]
11 Function CalcFreeness(instance):
12     if instance.isTerminating then
13         AddFakeReq (instance.requests)
14     totalVirtualUsages = 0
15     for req in instance.requests do
16         totalVirtualUsages+ =CalcVirtualUsage(req, instance)
17     freeness = (instance.M − totalVirtualUsage)/instance.B
18     return freeness
```

Virtual Usage + Freeness Algorithm

Calculates 'freeness' of a given instance

Algorithm 1: Virtual Usage and Freeness Calculation

```
1  Function CalcVirtualUsage(req, instance):
2      if req.isQueuing then
3          if req.isHeadOfLine then
4              return req.demand
5          return 0
6      if req.isFake then
7          return ∞
8      return req.physicalUsage+GetHeadroom(req.priority, instance)
9  Function GetHeadroom(p, instance):
10     return headroomForPriority[p]/instance.numRequests[p]
11 Function CalcFreeness(instance):
12     if instance.isTerminating then
13         AddFakeReq (instance.requests)
14     totalVirtualUsages = 0
15     for req in instance.requests do
16         totalVirtualUsages+ =CalcVirtualUsage(req, instance)
17     freeness = (instance.M − totalVirtualUsage)/instance.B
18     return freeness
```

Experimental Setup

Purpose: Evaluate KV-Cache Transfer on real hardware and integrate results with simulator

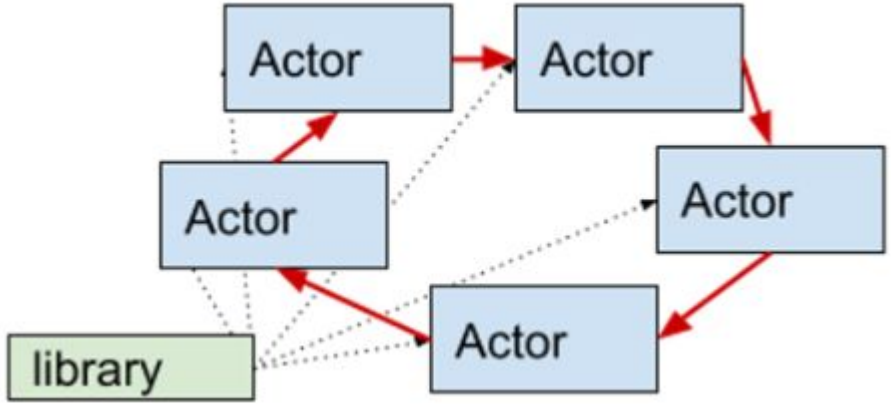
- On top of vLLM
  - Run on 2 DGX-H100s (Splitwise-HH) and 2 DGX-A100s (Splitwise-AA)
- Mixed Continuous Batching

KV-Cache Transfer Mechanism (for both naive and layer-wise)

- MSCCL++ library for communication (one-sided put primitive)
- Semaphores used for precise synchronization

Implementation

- Uses Ray actors to host each instance
- Communication between actors/instances through Gloo
- Frontend actors as API endpoints to handle user requests



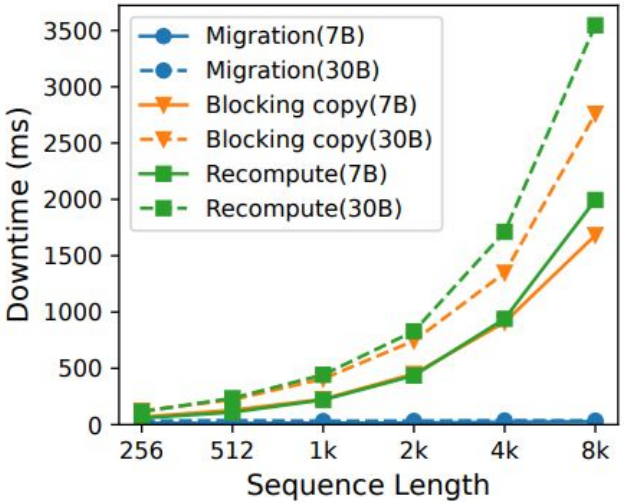
- Block fusion - copies blocks to contiguous buffer
- Fault tolerance - if global scheduler fails → simple scheduling

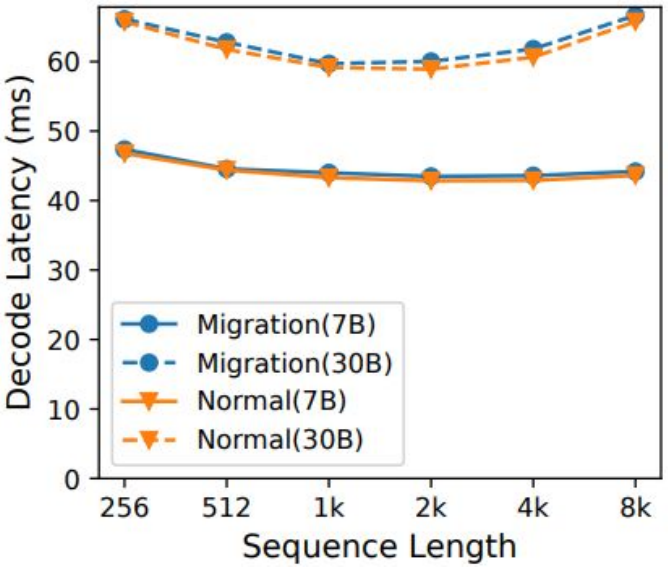
Evaluation Results

Compared to dispatch-time load balancing (INFaaS)

- Up to 2.2x/5.5x for first-token (mean/P99)
- Up to 1.3x for per-token generation P99
- 70% less preemption loss -> 1.3 second decrease in e2e latency
- 92% reduction of memory fragmentation

Migration Efficiency





- Downtime of baseline increase with sequence length reaching 111x of migration
- Example: 8k sequence
  - Recompute: 3.5 s ~ 54 decode steps
  - Migration: 2 steps

- Up to 1% performance difference with migration
- Fraction of time spent during migration was 10%

Memory Fragmentation Evaluation

- Fragmentation Proportion: percent of memory that could fit more requests if on a singular instance

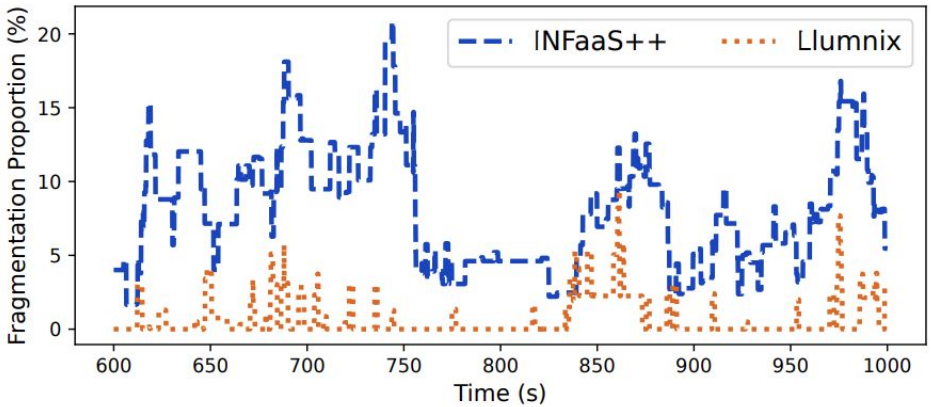


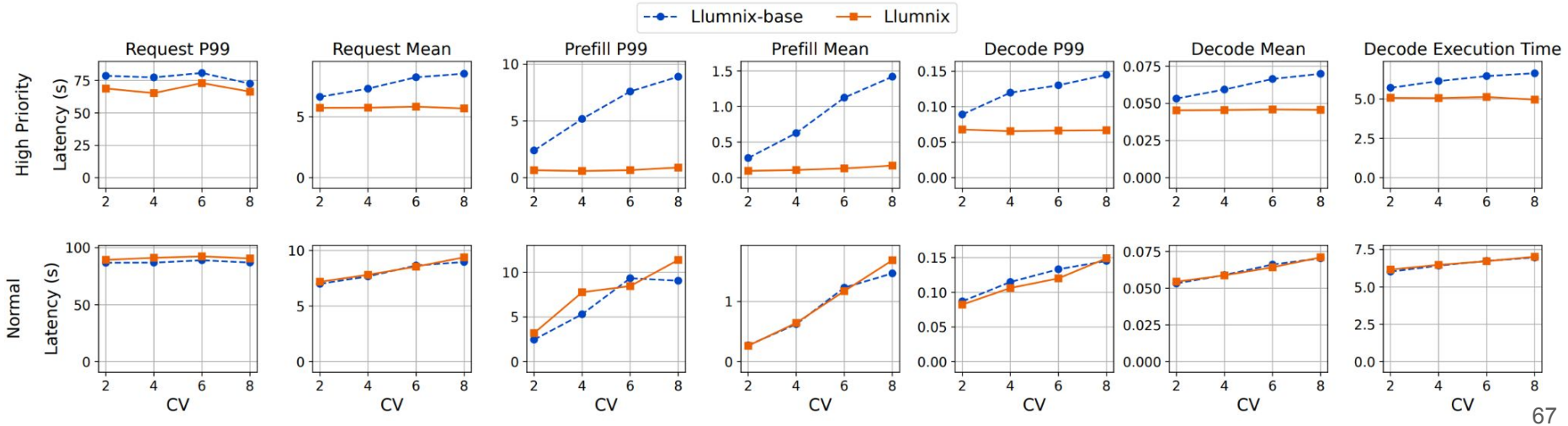
Figure 12: Memory fragmentation over time.

70% less preemption loss -> 1.3 second decrease in e2e latency

92% reduction of memory fragmentation

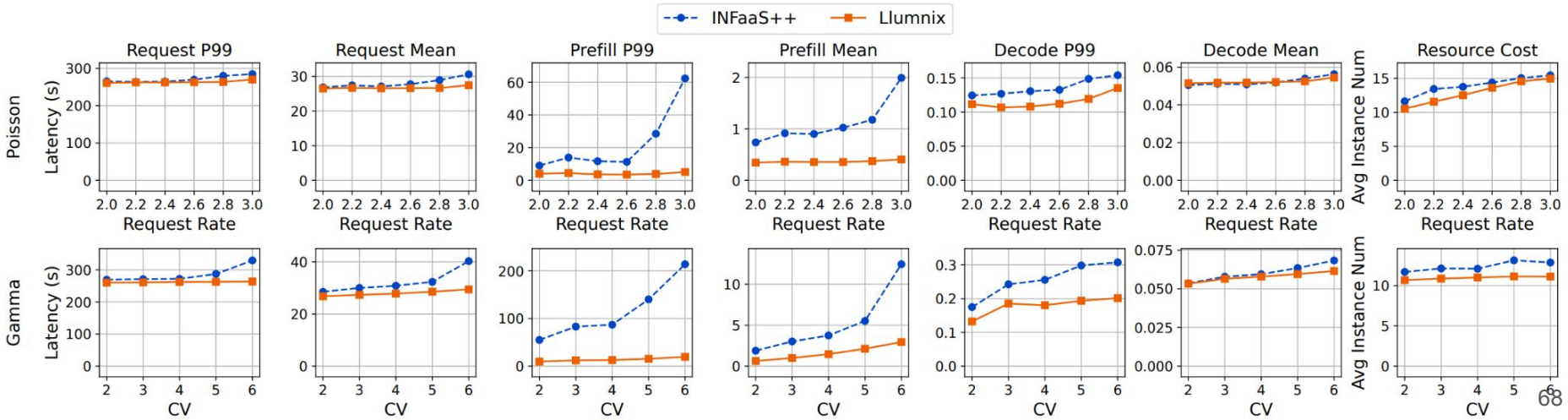
Priority Requests Evaluation

- For high priority requests:
  - 1.2-1.5x decrease in request latencies
  - 2.6-8.6x (mean) and 3.6-10x (p99) decrease for prefill latencies
  - 1.2x-1.5x (mean) and 1.3x-2.2x (p99) decrease for decode latencies



Auto-Scaling Evaluation

- Poisson:
  - 12.2x for prefill latency
  - 16% costs savings
- Gamma
  - 11x for prefill latency
  - 18% cost savings



Simulator Setup

Purpose: Explore cluster designs and evaluate Splitwise at scale

1. Profile LLM on target hardware
2. Build performance model based on characterization profiles & feed in orchestration and application inputs
3. Simulator provides achieved metrics
4. Cross-validate performance model with hardware experiments
5. E2E evaluation using production load

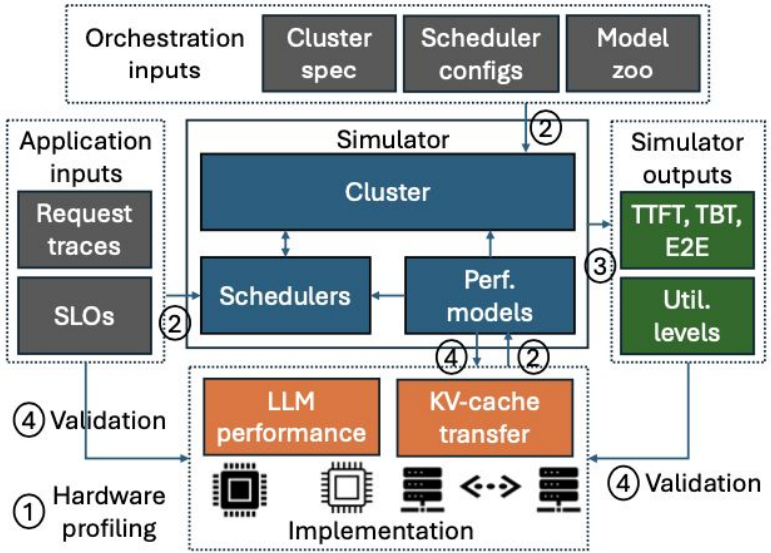
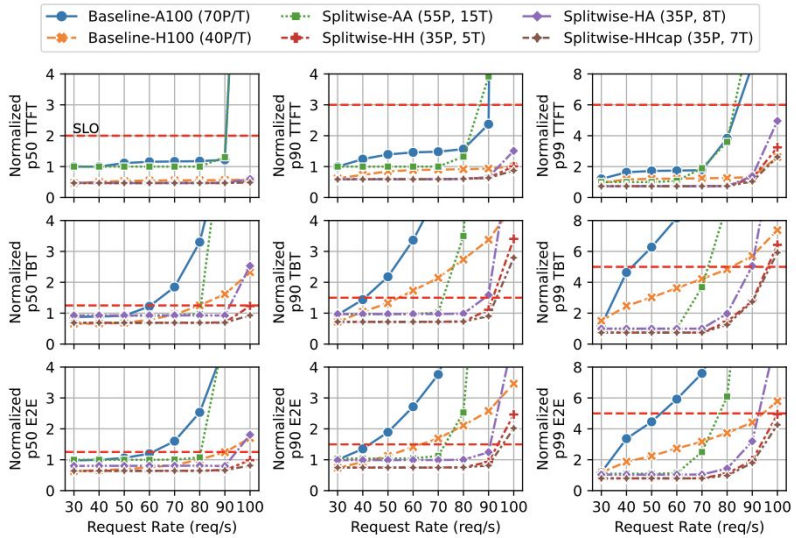


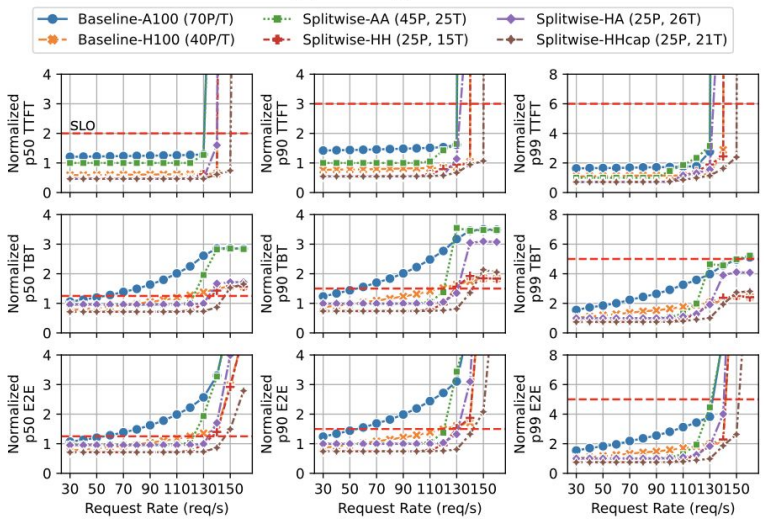
Fig. 13: Overview of the design of the Splitwise simulator.

Evaluation – Iso-power throughput-optimized clusters

- Explores **maximizing throughput** while keeping the total power consumption of the cluster **constant**
- Different provisioning choices under **coding** and **conversation** workloads



(a) Coding trace.

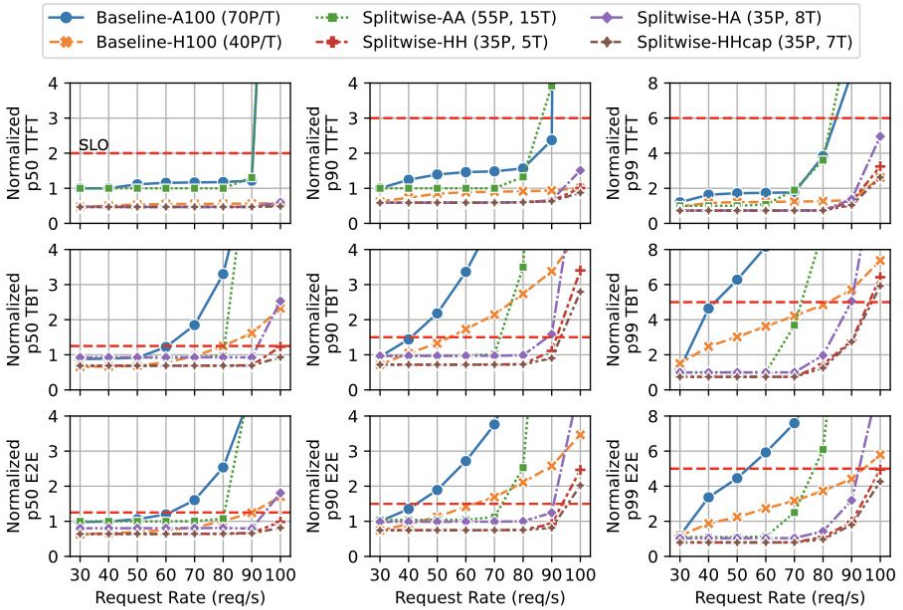


(b) Conversation trace.

Evaluation – Iso-power throughput-optimized clusters

Coding Workload

- Splitwise-HH, Splitwise-AA, Splitwise-HHcap perform better than Baseline-H100
- Splitwise-AA has high TTFT
- Splitwise-HA bridges gap with low TTFT and E2E at high throughput
- Splitwise overall more useful at higher loads because of mixed machine pool

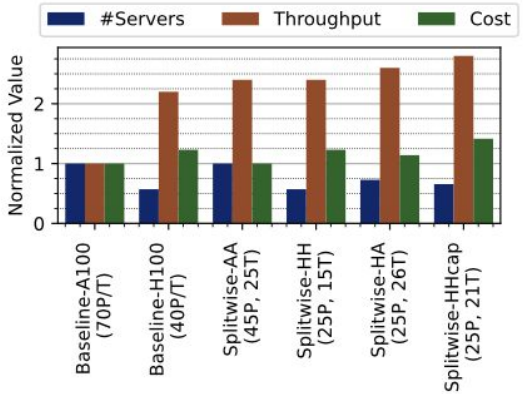


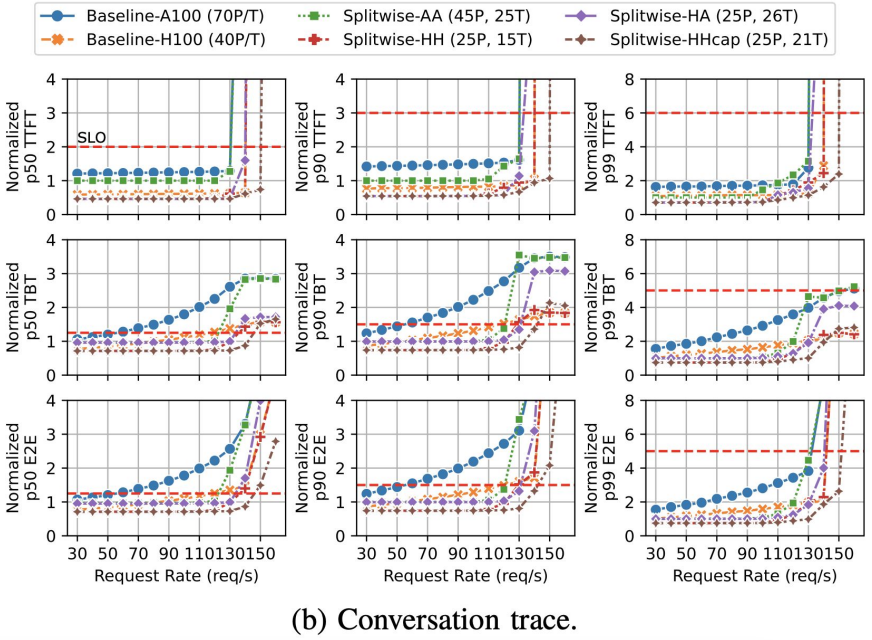
(a) Coding trace.

Evaluation – Iso-power throughput-optimized clusters

Conversation Workload

- Splitwise-HHcap excels across all latency metrics
- Demonstrates benefits of power capping for token generation phase





(b) Conversation trace.

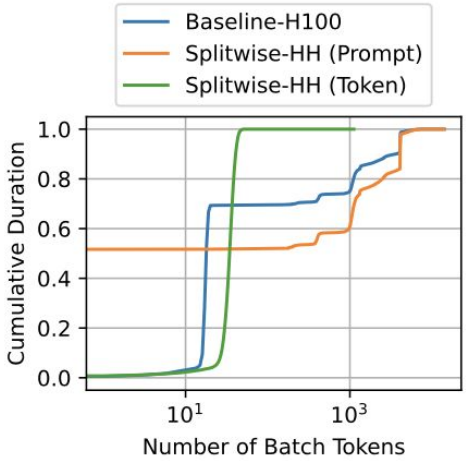
Evaluation – Impact on Batched Tokens

More efficient batching at low loads

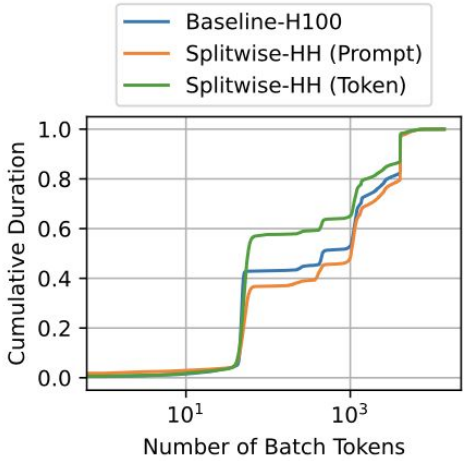
- Baseline-H100 spends 70% of time running < 15 tokens (smaller batch sizes)
- Splitwise-HH prompt mostly idle but runs much larger batches of tokens when active
- Splitwise-HH token also do better job at batching

Perform similar at higher loads due to mixed pool

- Machines handle both prompt and token processing



(a) Low load (70 RPS).



(b) High load (130 RPS).

Evaluation – Impact of workload changes

So far, we have tested a trace and a model on clusters optimized for a specific workload pattern and model

Want to test robustness by running conversation trace on coding service cluster & Llama-70B on a cluster meant for BLOOM-176B

Changing Workload Trace:

- Splitwise-HA & Splitwise-HHcap experience 7% throughput setback

Changing Model:

- Llama-70B has fewer parameters so can support higher throughput on BLOOM-176B cluster design
- Splitwise designs outperform baselines at higher loads

Dynamic LLM Workloads

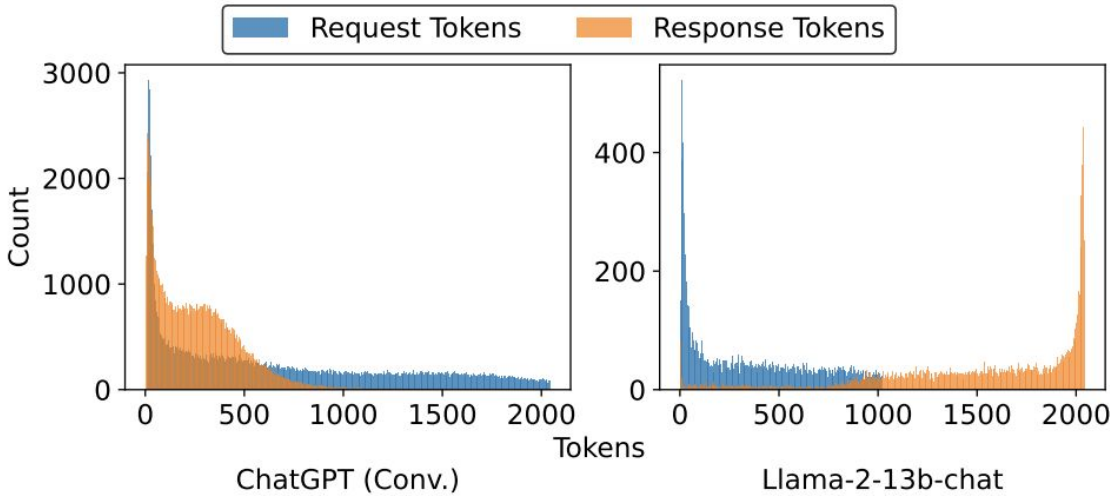


Figure 8: Distribution of Request and Response Tokens.

What leads to inefficient memory and resource allocation in LLMs?

How can we use workload characteristics to tailor resource allocation and reduce fragmentation?

Cite: https://arxiv.org/pdf/2401.17644v3

Related Works

Heterogeneous Scheduling and Dataflow Systems

- Prior work on heterogeneous scheduling explores balancing cost, energy, and performance on single machines
- Research on multiprocessor CPU scheduling allocates workloads based on request length and hardware performance counters

Splitwise Design Overview

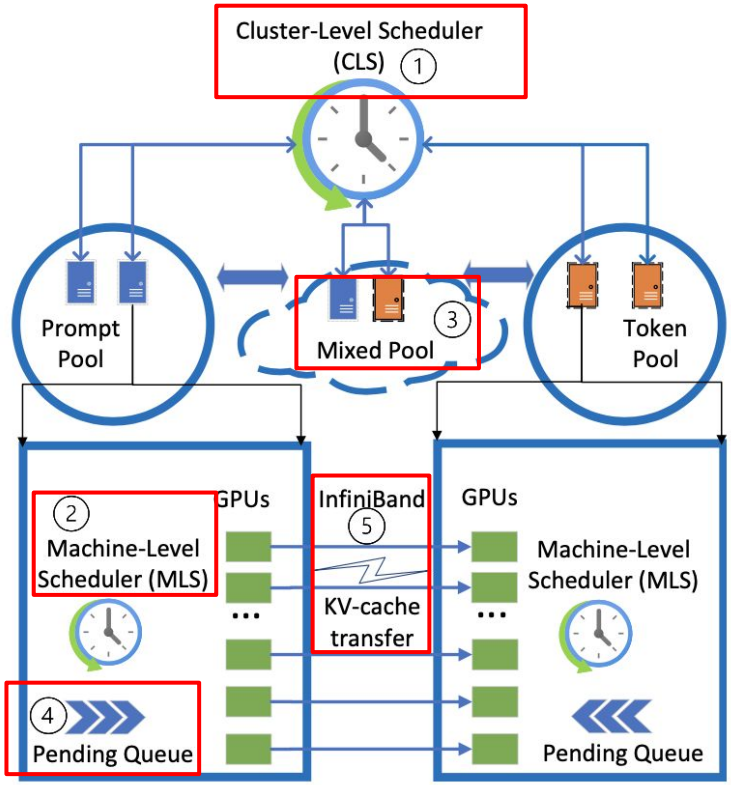


Fig. 10: High-level system diagram of Splitwise.

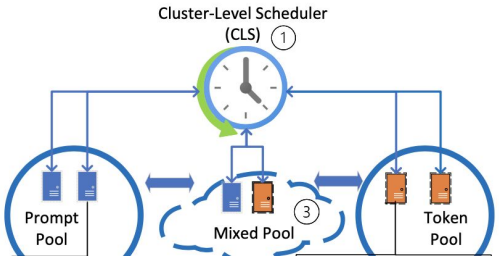
Cluster Level & Machine Level Schedulers

CLS

Key Function: Manages resource allocation across machine pools

How it works:

- Join Shortest Queue for prompt/token machine assignment.
- Dynamic resource adjustment; switch to/from mixed pool
- Efficient KV-cache transfer with prompt/token pairing

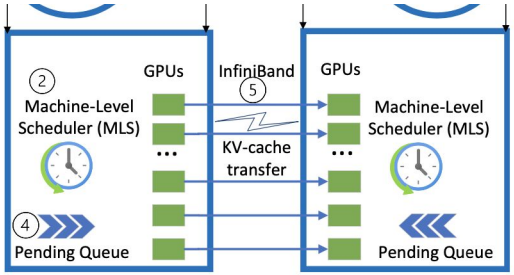


MLS

Key Function: Manages task queues, batching, mem for each machine.

How it works:

- **Prompt Machines:** First-Come-First-Serve Scheduling, token limit
- **Token Machines:** Batches tokens until memory full
- **Mixed Machines:** Prioritize prompts for SLO, preempting tokens

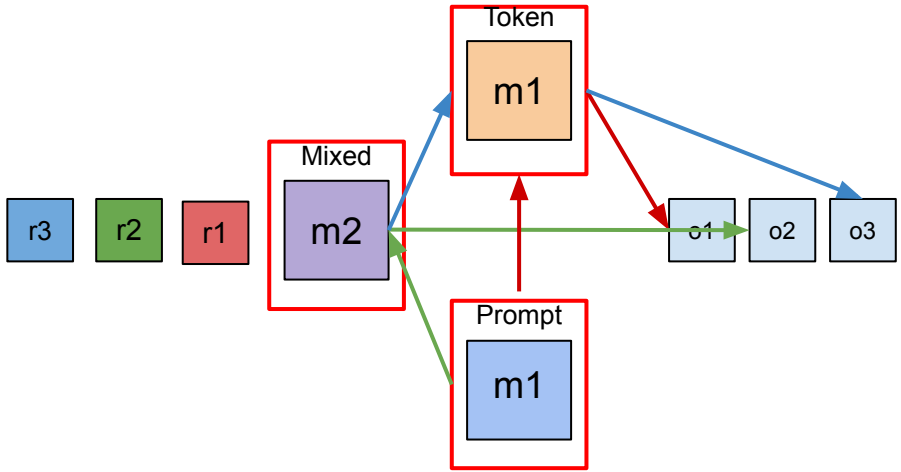


Mixed Machine Pool

Purpose: Balance both prompt and token tasks dynamically, based on real-time workload demands

How it works:

- Prioritize prompt tasks for SLOs
- Shares resources for prompt and token phases
- Increase priority of preempted token phases with age to avoid starvation



Provisioning With Splitwise

Number of Machines

- Cluster deployment must be sized with appropriate number of prompt and token machines
- Search design space using event-driven cluster simulator

Setup

Experiment

Evaluate KV-Cache Transfer on real hardware

- 2 H100s & 2 A100s
- MSCCL++ library for communication (one-sided put primitive)
- Semaphores used for precise synchronization

Simulator

Explore cluster designs and evaluate Splitwise at scale

- Evaluated across two production traces: conversation and coding workloads