Mooncake

A KVCache-centric Disaggregated Architecture for LLM Serving (FAST ‘25)

Ruoyu Qin, Zheming Li, Weiran He
Mingxing Zhang, YongweiWu, Weimin Zheng, Xinran Xu,
Moonshot AI | Tsinghua University

Context: What is Mooncake?

Moonshot AI & Kimi

> Moonshot AI: Chinese AI startup, similar to Anthropic or OpenAI

> Kimi: their flagship LLM product, like ChatGPT or Claude

> Specialises in very long context windows (128k+ tokens)

> Average input length: 7,590 tokens; output: 182 tokens

> Primary use: users send large documents for summarisation

> Mooncake is the serving infrastructure that powers Kimi

What is LLM Serving?

> Running a trained LLM as a cloud service over HTTP

> User sends prompt, service returns generated tokens

> Your phone app is just a UI, all compute is in a datacenter

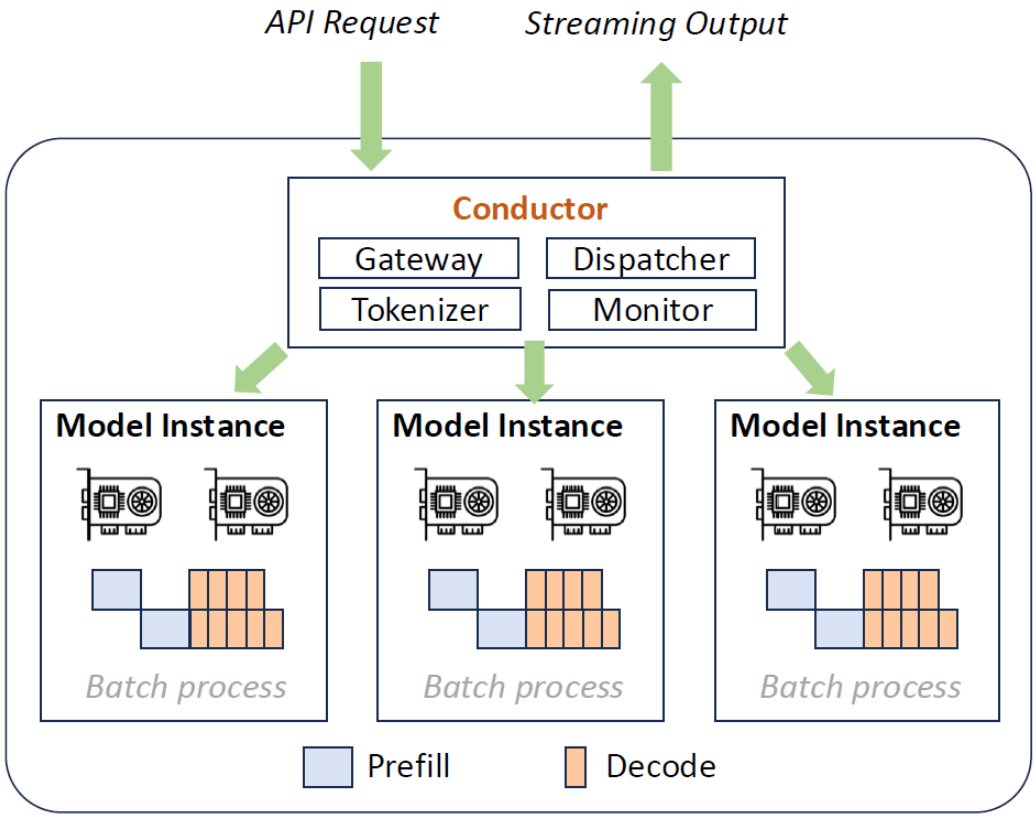
> Model weights (140 GB for 70B) live on expensive GPU clusters

> Goal: maximise throughput while meeting latency guarantees

> Examples: vLLM, FasterTransformer, TensorRT-LLM, Orca

Mooncake is the serving infrastructure that makes Kimi fast and cost-effective at scale, handling 75% more real-world requests vs previous systems.

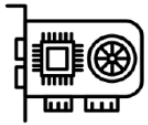
Online LLM Serving System



Online LLM Serving System

More Data + Larger Model + Longer Context = ***Higher Intelligence***

BUT



Lack of GPU Supply

Higer Inference Cost

Longer Response Time

Online LLM Serving System

More Data + Larger Model + Longer Context = ☺ ***Higher Intelligence***

Reduce the cost for large-scale long-context LLM inference!

Lack of
GPU Supply

Higer
Inference Cost

Longer
Response Time

LLM Inference: Two Distinct Phases

PREFILL PHASE [Compute-Bound]

> Process ALL input tokens simultaneously

> One big matrix-matrix multiply (GEMM)

> Produces KV cache for every input token

> Generates the very first output token

> Scales quadratically with sequence length

> GPU arithmetic fully saturated

> **Bottleneck: GPU FLOPS -> better GPU = faster prefill**

DECODING PHASE [Memory-Bound]

> Generates ONE token per step

> Autoregressive -> cannot parallelise output tokens

> Reads ENTIRE KV cache from VRAM every step

> Tiny compute but massive memory reads

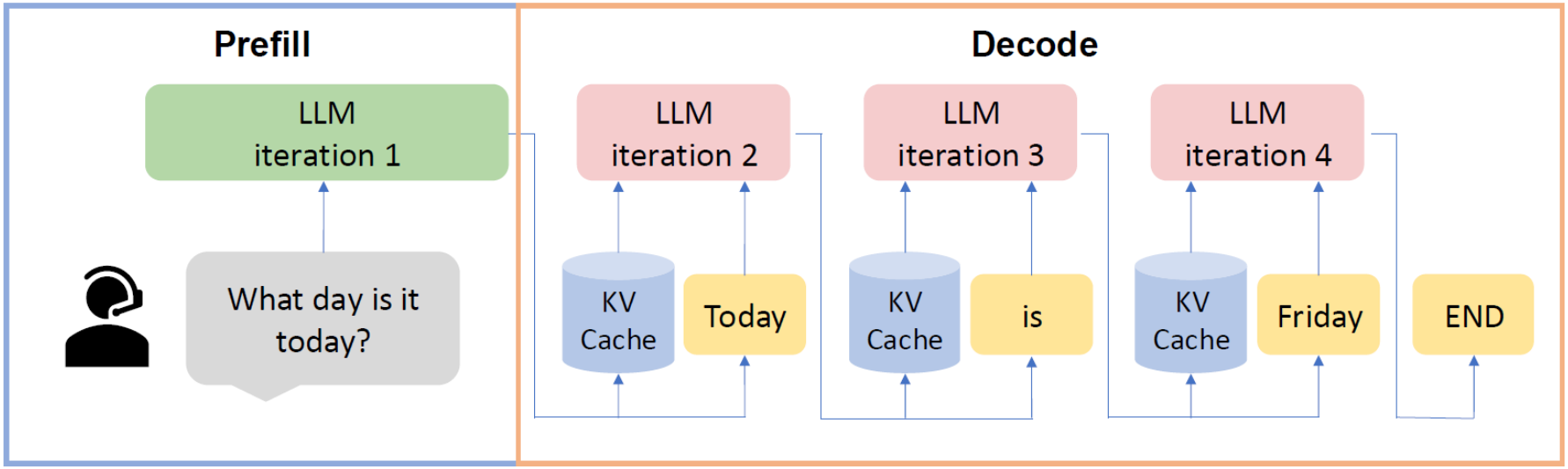
> GPU cores idle, waiting for data from VRAM

> Bottleneck: VRAM bandwidth

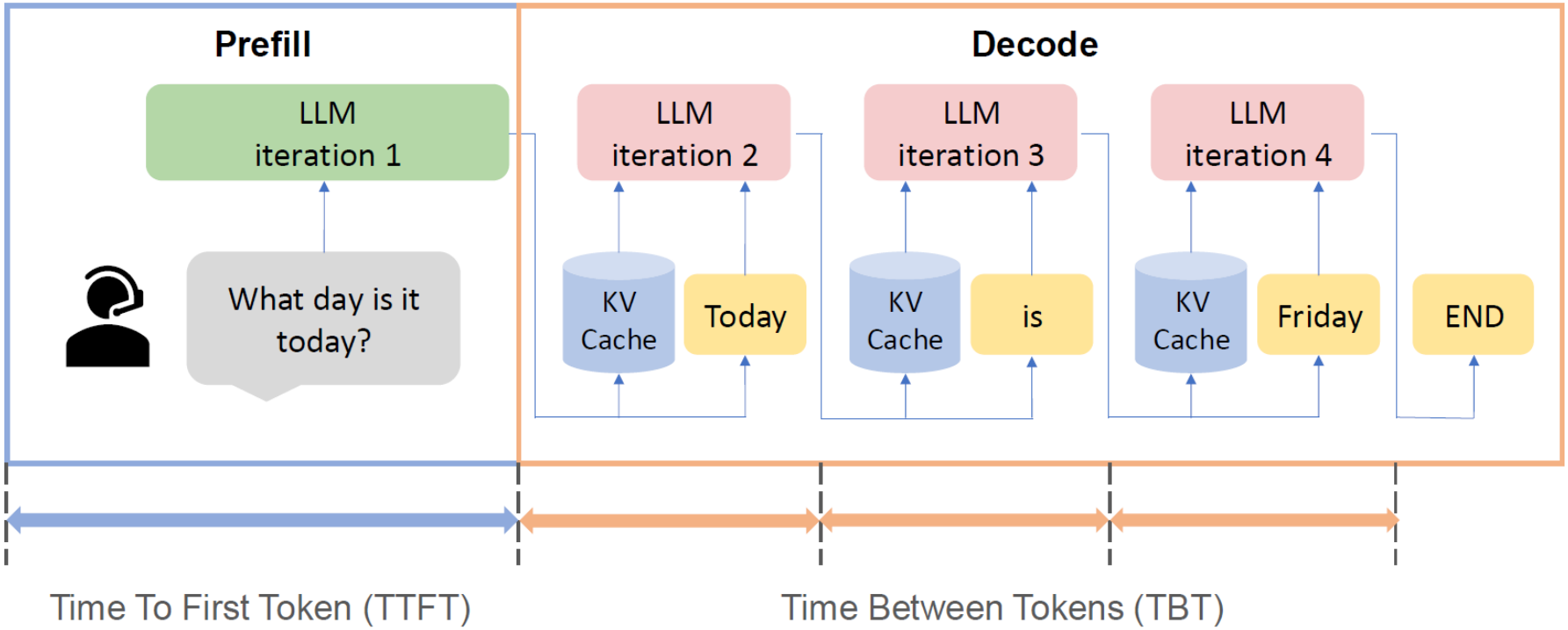
> **More memory bandwidth = faster decoding**

Key insight: these phases have fundamentally different bottlenecks -> they should run on different hardware (disaggregation).

LLM Inference



LLM Inference



Service Level Objectives (SLOs)

TTFT -- Time to First Token

Delay from user sending request to first response word appearing. Measures prefill latency. Users experience this as 'responsiveness'.

Example SLO: TTFT_P90 = 10x means 90% of requests must finish within 10x the unloaded single-request latency.

TBT -- Time Between Tokens

Time between each successive token streaming to the user. Measures decoding per-step latency. Large TBT means text dribbles out slowly.

Example SLO: TBT_P90 = 5x means 90% of tokens must appear within 5x the unloaded per-step latency.

Goodput -- The Key Metric

Only requests that FULLY COMPLETE within SLO count toward throughput. If a request is rejected by the decoding node after prefill has already completed, ALL that prefill compute is wasted -- zero contribution to goodput. This shapes every design decision in Mooncake: waste must be prevented proactively.

LLM Inference: Latency

	TTFT	TBT
Program (~1k tokens)	Low (~100ms)	Very low (~50ms)
Web search (~8k tokens)	High (~1s)	Match read speed (~100ms)
Multi-document summary (~128k tokens)	Very high (> 5s)	Match read speed (~100ms)

Why Existing Systems (vLLM) Fail at Scale

Problem 1: Prefill-Decoding Interference

In vLLM, prefill and decoding share the same GPU. A 100k-token prefill monopolises the GPU for several seconds. All concurrent decoding requests are stalled, TBT SLO violated for every user in the batch during that window.

Problem 2: GPU VRAM Bottleneck

A 128k token request needs ~40 GB of KV cache alone, more than half an A100. After loading model weights, only ~10-12 long-context requests fill all available VRAM, forcing tiny batch sizes and collapsing GPU utilization during decoding.

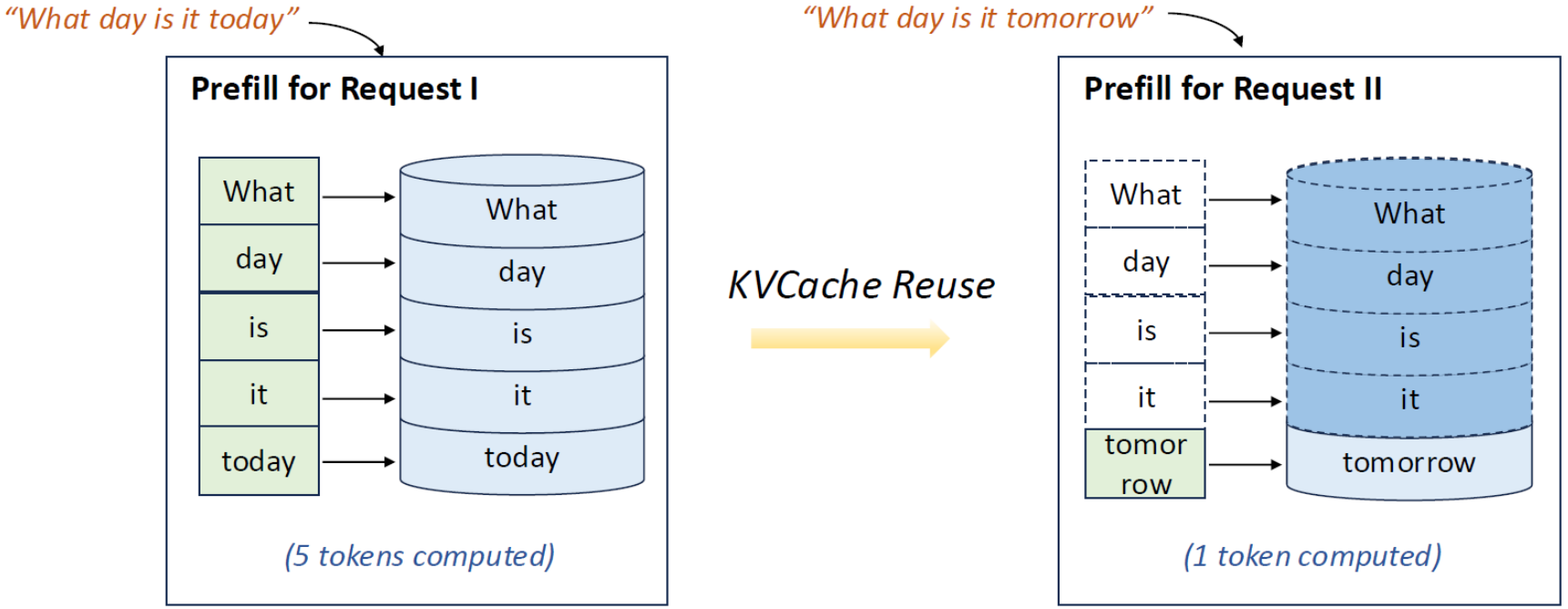
Problem 3: KV Reuse is Node-local and VRAM-bound

vLLM caches KV prefixes but only within a single node's VRAM, which gets evicted under load and is invisible to other nodes. When 1,000 users share a 50k-token system prompt, every node recomputes it independently. There is no cluster-wide mechanism to share cached KV across machines.

Root cause: GPU memory is the bottleneck and none of these systems manage it intelligently.

LLM Inference: Prefix Caching

- KVCache can be shared across requests with the same prefix, reducing computation during prefill

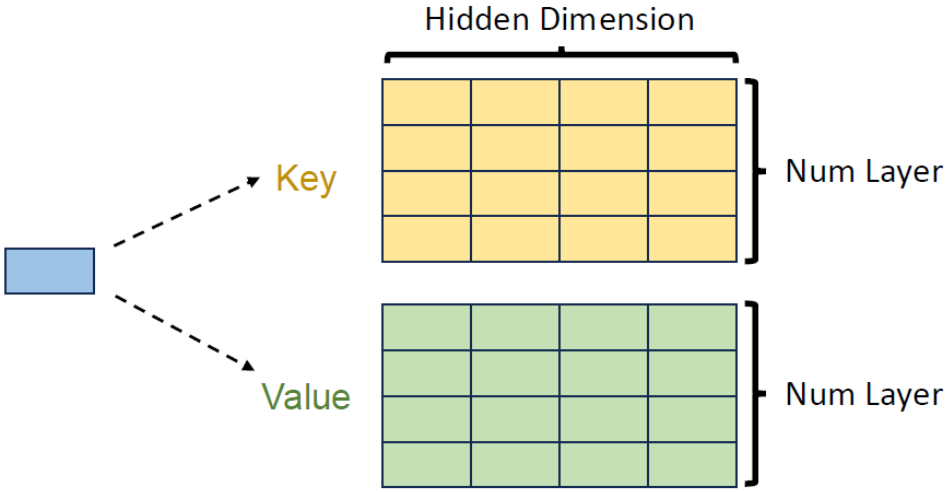


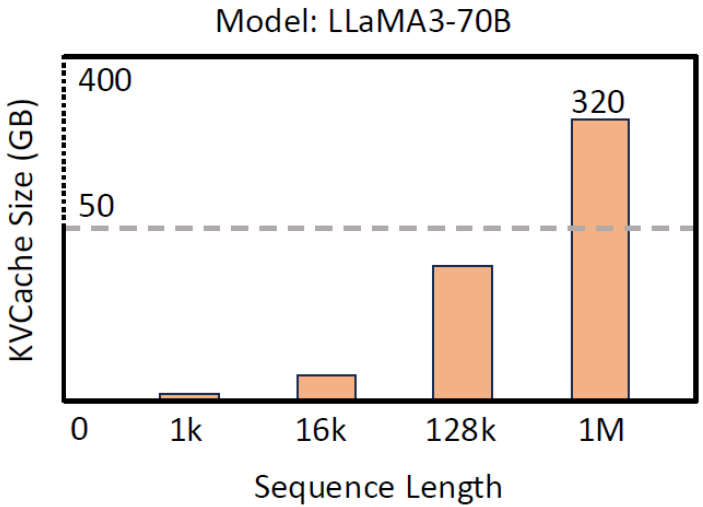
KVCache: Huge Challenge to Storage System

- One token's KVCache size is quite large
- KVCache linearly increases with the sequence length

One token (Bytes)

One token's KVCache (Tens of KB)





KVCache: Huge Challenge to Storage System

- The volume of reusable KVCache is much larger than the available storage capacity of a single inference node

Over **100 billion** tokens per day

One Token
(Bytes)

One Token's
KVCache
(Tens of KB)

GPU VRAM
(Hundreds of GB)

DRAM or Other
Storage (TB)

Reusable KVCache (Hundreds of TB ~ PB)

Trading More Storage for Less Computation

Requirement 1

Large-capacity KVCache storage

Requirement 2

Low-latency and high-bandwidth KVCache transfer

Requirement 3

KVCache-aware scheduling

Mooncake's Three Core Ideas

1. Disaggregation

> Physically separate prefill and decoding onto different GPU servers

> No more interference, each of it is optimised independently

> Prefill nodes: optimised for compute-intensive parallel processing

> Decoding nodes: optimised for memory-bandwidth efficiency

> KV cache is transferred between nodes via RDMA after prefill

2. KVCache-Centric Design

> KV cache management is the central scheduling problem

> Every routing decision is driven by KV cache state

> Reuse cached KV to skip redundant prefill computation

> Schedule requests to nodes that already have matching cache

> Hot cache blocks replicated automatically across cluster

3. Distributed Cache Pool

> Aggregate all underutilised CPU DRAM + SSD cluster-wide

> Much larger than GPU VRAM alone (100 nodes x 512 GB = 50 TB)

> Hash-based deduplication identifies reusable KV blocks

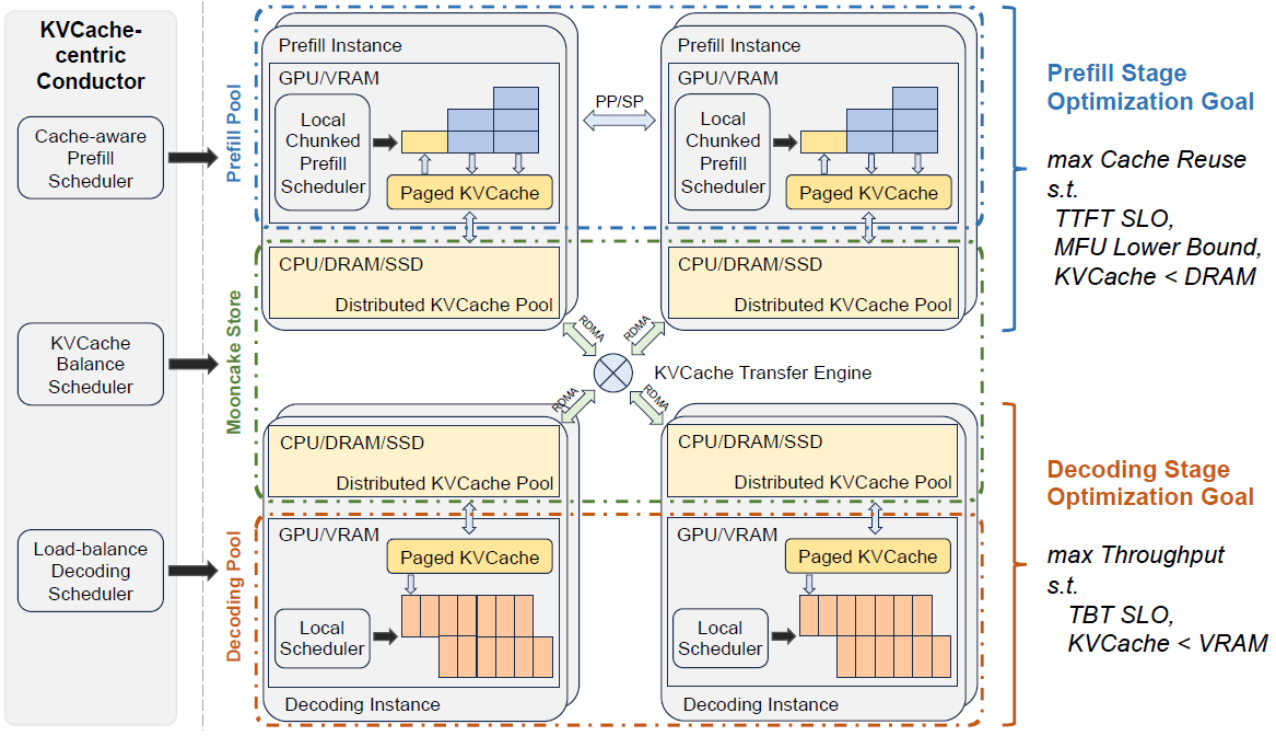
> LRU/LFU eviction policies manage capacity

> Near-GPU prefix caching at low additional cost

Together, these three ideas enable Mooncake to serve 525% more throughput than vLLM for 128k-token contexts by eliminating interference, fragmentation, and redundant computation.

Mooncake: A KVCache-centric Disaggregated Architecture

- P&D disaggregation architecture centered around the distributed KVCache pool – Mooncake Store



Mooncake Disaggregated Architecture Overview

Conductor

- Global scheduler
- Cache-aware prefill scheduler
- KVCache balance scheduler
- Load-balance decoding scheduler
- Early rejection policy
- Predicts future load

Prefill Pool

- GPU/VRAM -> runs prefill computation
- CPU DRAM -> KV cache staging & pool
- Prefix KV reuse from distributed pool
- CPP for long-context multi-node prefill
- Layer-wise KV streaming to decode nodes

Decoding Pool

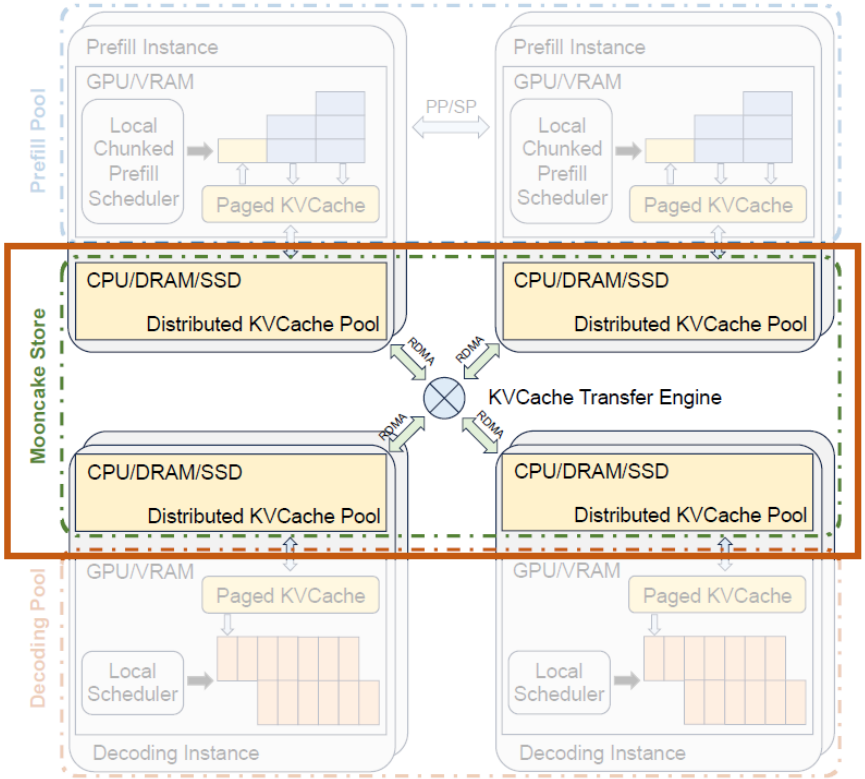
- GPU/VRAM -> continuous batching
- CPU DRAM -> async KV cache loading
- PagedAttention for memory efficiency
- TBT-SLO based load balancing
- Multiple requests per GPU batch

Distributed KV Pool

- GPU VRAM: active computation
- CPU DRAM: warm KV cache
- SSD: cold KV cache
- Aggregated across all cluster nodes
- Hash-based block deduplication
- LRU / LFU eviction
- GPUDirect RDMA transfer
- ~51% cache hit rate (real data)

Mooncake Store – Distributed Multi-layer KVCache Pool

- Large size and bandwidth (**Key of KVCache Storage**)
  - Distributed KVCache pool across all inference nodes
  - Utilize high performance connection like (GPUDirect) RDMA/Storage
  - Optimized for multi-NIC scenarios
- Design features
  - High scalability and fault tolerance
  - Independent to specific inference engine
  - Flexible API (adopted by vLLM [1])



Distributed KV Cache Pool & Hash-based Deduplication

Memory Hierarchy

GPU VRAM

Bandwidth: ~2,000 GB/s

Capacity: 80 GB / node

Active computation -- KV must be here during attention

CPU DRAM

Bandwidth: ~150 GB/s

Capacity: 512 GB - 2 TB / node

Warm KV cache pool, staging for transfers

SSD

Bandwidth: ~7 GB/s

Capacity: TBs / node

Cold KV cache (rarely accessed blocks)

Hash-based Block Deduplication

Tokens grouped into fixed-size blocks (e.g. 512 tokens). Each block gets a prefix-chained hash:

```
Hash(block 0) = Hash(tokens 0-511)
```

```
Hash(block 1) = Hash(H0 + tokens 512-1023)
```

```
Hash(block N) = Hash(H_prev + tokens ...)
```

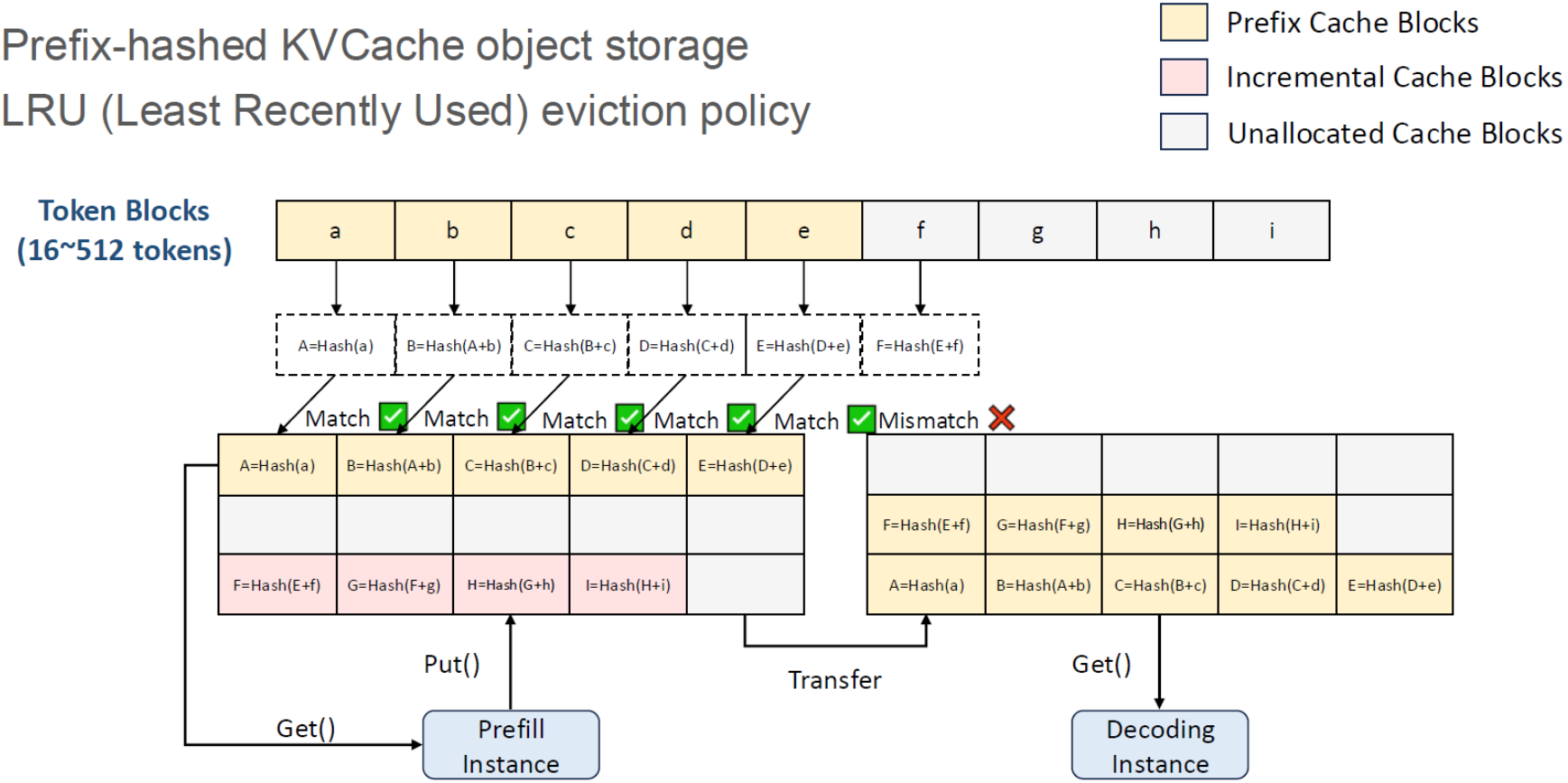
Same hash = same tokens + same prefix = KV is reusable.
Hash computed from token IDs only -> no GPU compute needed.
Checked in order from first block until first mismatch.

Example:

1,000 users share a 2,000-token system prompt. KV computed once, cached in DRAM pool. All others skip that prefill entirely -- just load from DRAM.

Mooncake Store: KVCache Management

- Prefix-hashed KVCache object storage
- LRU (Least Recently Used) eviction policy



4-Step Request Workflow

1 KVCache Reuse

Conductor hashes prompt tokens. Searches distributed DRAM pool for matching blocks. Finds longest prefix match. Prefill node loads cached KV blocks (CPU DRAM to GPU VRAM).

2 Incremental Prefill

Prefill node computes KV only for UNCACHED tokens. If uncached portion exceeds threshold, split into chunks processed via Chunked Pipeline Parallelism across multiple nodes.

3 KVCache Transfer

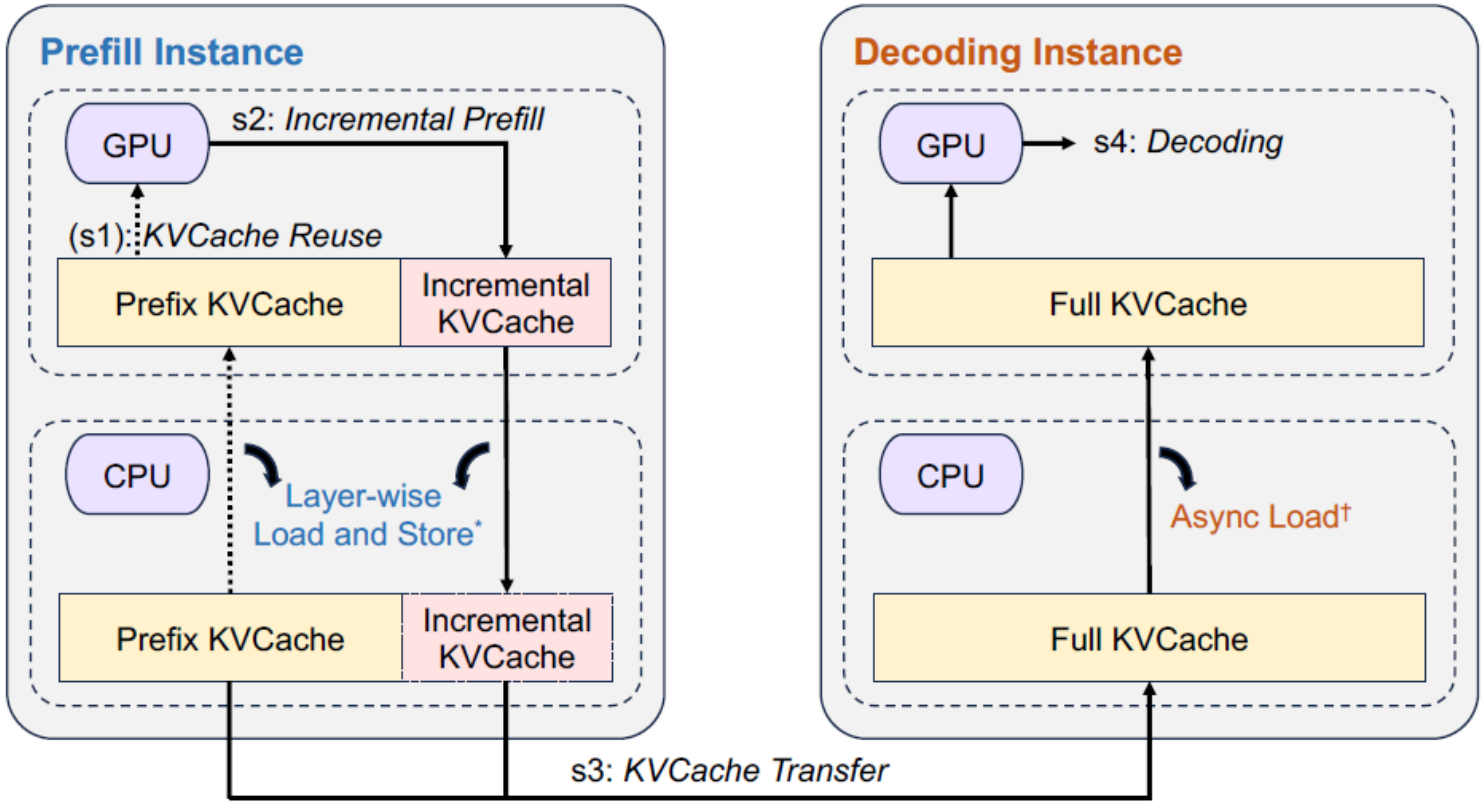
Messenger uses GPUDirect RDMA to stream new KV from prefill node to decoding node CPU DRAM. Asynchronous -> overlapped with Step 2 layer-by-layer. Up to 800 Gbps.

4 Decoding

Decoding node loads full KV from CPU DRAM to GPU VRAM. Request joins continuous batching batch. Generates tokens one by one. Local scheduler checks TBT SLO before accepting.

KVCache Reuse — **Incremental Prefill** — **KVCache Transfer** — **Decoding**

Each step reduces latency: Reuse avoids prefill entirely. Incremental prefill skips cached tokens. Overlapped transfer hidesnetwork cost. TBT check prevents wasted compute.


Prefill Instance
GPU
s2: Incremental Prefill
(s1): KVCache Reuse
Prefix KVCache
Incremental KVCache
CPU
Layer-wise Load and Store*
Prefix KVCache
Incremental KVCache
Decoding Instance
GPU
s4: Decoding
Full KVCache
CPU
Async Load†
Full KVCache
s3: KVCache Transfer

Chunked Pipeline Parallelism (CPP): Multi-node Prefill

Problem: A 128k-token prefill cannot fit on one GPU. Multiple nodes must cooperate -> but HOW they communicate is critical. CPP is Mooncake's approach.

Tensor Parallelism (TP)

Comm: All-reduce after every layer (160+ comms for 80-layer model)

+ Works within one server on NVLink

- Cross-node all-reduce severely hurts GPU utilisation (MFU)

Sequence Parallelism (SP)

Comm: Ring comm inside attention at every layer, competes with KV transfers

+ Distributes sequence tokens across nodes

- Frequent blocking communications, competes with network bandwidth

CPP -- Mooncake's Choice

Comm: Only at pipeline stage boundaries, overlapped with GPU computation

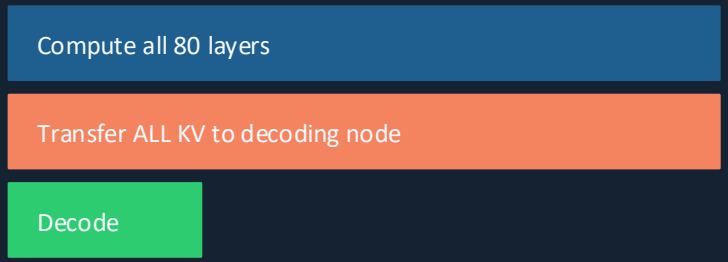
+ High MFU, unidirectional, overlapped with compute, no blocking

- Requires pipeline coordination between prefill nodes

CPP: chunks processed simultaneously in assembly-line fashion. Communication only at stage boundaries -> overlapped with GPU compute.

Layer-wise Prefill: Overlapping KV Transfer with Computation

WITHOUT Layer-wise Prefill



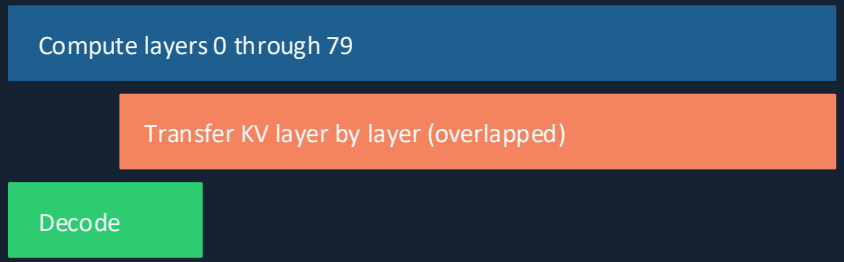
TTFT = T_compute + T_transfer

Sequential: transfer adds full cost to the critical path.

What happens:

All 80 layers computed first. Entire KV cache held in VRAM simultaneously. After all layers done, full transfer begins. GPU idles during transfer. For 128k tokens this adds significant latency.

WITH Layer-wise Prefill



TTFT ≈ max(T_compute, T_transfer)

Transfer is hidden inside the pipeline so it's approx free!

How it works:

After layer N attention: async store KV_N to DRAM + async RDMA transfer to decoding node. Immediately start layer N+1. VRAM holds only 1-2 layers KV at a time -> massive VRAM saving for long contexts.

KVCache-centric Scheduling Algorithm (Conductor)

For every incoming request R, Conductor selects optimal prefill node p and decoding node d:

Step 1	Compute prefix-chained block hashes for R's prompt tokens to get block_keys
Step 2	Search all prefill instances for longest prefix match: best_prefix_len, best_matched_instance
Step 3	For each instance estimate TTFT = T_queue + T_transfer (if remote cache needed) + T_prefill (uncached tokens)
Step 4	Select decoding instance d with lowest current batch load to meet TBT SLO
Step 5	If TTFT > TTFT_SLO or TBT > TBT_SLO: REJECT immediately (HTTP 429). Zero compute wasted.
Step 6	If best remote prefix significantly exceeds local prefix: trigger KVCache hot-spot migration (proactive block replication)

Algorithm 1 KVCache-centric Scheduling Algorithm

Input: prefill instance pool P, decoding instance pool D, request R, cache block size B.
Output: the prefill and decoding instances (p, d) to process R.

$block\_keys \leftarrow \text{PrefixHash}(R.prompt\_tokens, B)$
$TTFT \leftarrow \text{inf}$
$p \leftarrow \emptyset$
$best\_prefix\_len, best\_matched\_instance \leftarrow \text{FindBestPrefixMatch}(P, block\_keys)$
for $instance \in P$ **do**
    $prefix\_len \leftarrow instance.prefix\_len$
    $T_{queue} \leftarrow \text{EstimatePrefillQueueTime}(instance)$
    if $\frac{best\_prefix\_len}{prefix\_len} <$ **kvcache_balancing_threshold then** ▷ Cache-aware prefill scheduling
        $T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt\_tokens), prefix\_len)$
        if $TTFT > T_{queue} + T_{prefill}$ **then**
            $TTFT \leftarrow T_{queue} + T_{prefill}$
            $p \leftarrow instance$
        end if
    else ▷ Cache-aware and -balancing prefill scheduling
        $transfer\_len \leftarrow best\_prefix\_len - prefix\_len$
        $T_{transfer} \leftarrow \text{EstimateKVCacheTransferTime}(instance, best\_matched\_instance, transfer\_len)$
        $T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt\_tokens), best\_prefix\_len)$
        if $TTFT > T_{transfer} + T_{queue} + T_{prefill}$ **then**
            $TTFT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$
            $p \leftarrow instance$
        end if
    end if
end for
$d, TBT \leftarrow \text{SelectDecodingInstance}(D)$ ▷ Load-balancing decoding scheduling
if $TTFT > TTFT\_SLO$ **or** $TBT > TBT\_SLO$ **then**
    reject R; **return**
end if
if $\frac{best\_prefix\_len}{p.prefix\_len} >$ **kvcache_balancing_threshold then**
    $\text{TransferKVCache}(best\_matched\_instance, p)$ ▷ KVCache hot-spot migration
end if
return (p, d)

Cache Load Balancing & Automated Hot-spot Migration

The Hot-spot Problem

- Cache block popularity is wildly skewed
- Over 50% of cache blocks are NEVER accessed
- A few 'hot' blocks (popular system prompts) accessed tens of thousands of times
- One node holding a hot block gets overwhelmed with transfer requests
- Naive fix: global prediction model -> impossible, workload changes unpredictably

Demand-driven Hot-spot Migration

Trigger condition

best_prefix_len / local_prefix_len exceeds kvcache_balancing_threshold

Action taken

Tell node P to fetch hot blocks from best_matched_instance (node Q) via RDMA and store locally

Immediate effect

Current request uses transferred blocks. Future requests to P can use them locally so no transfer delay

Auto-replication

Hot blocks naturally replicate to all nodes that need them. Load distributes without a prediction model.

Blocks replicate where needed -> automatically, without a prediction model.

Overload-oriented Scheduling

Unlike academic work that assumes unlimited resources, Kimi is always overloaded at peak. GPU supply cannot keep up with user growth. Rejecting some requests is unavoidable -> the question is which ones, and when.

No Rejection

Pros:

+ All requests accepted

+ Simple to implement

Cons:

- SLOs violated for everyone

- System collapses under high load

Naive Early Rejection

Pros:

+ Prevents SLO violations

+ Reduces wasted prefill compute

Cons:

- Causes load oscillation between prefill and decoding pools

- Poor average cluster utilisation

Prediction-based Rejection

Pros:

+ Stable load across cluster

+ Minimal wasted compute

+ High goodput

Cons:

- Requires system-level load prediction model

- More complex implementation

Load Fluctuation Problem & Prediction-based Solution

Why Naive Early Rejection Oscillates

Stage 1

Both pools low load. Conductor accepts flood of requests. Prefill instances fill up.

Stage 2

Prefill completes. All requests hit decoding simultaneously. Decoding overloads. New requests rejected. Prefill goes idle.

Stage 3

No new prefills. Decoding drains. Both pools go idle despite available capacity.

Stage 4

Low decoding load detected. Conductor accepts another flood. Cycle repeats.

Result: prefill oscillates 20-90%, decoding anti-phase. Low cluster utilisation.

Prediction-based Early Rejection

Root cause: decisions based on CURRENT load but what matters is FUTURE load when this request's prefill completes.

System-level prediction:

> Assume each decoding request takes uniform time t_d

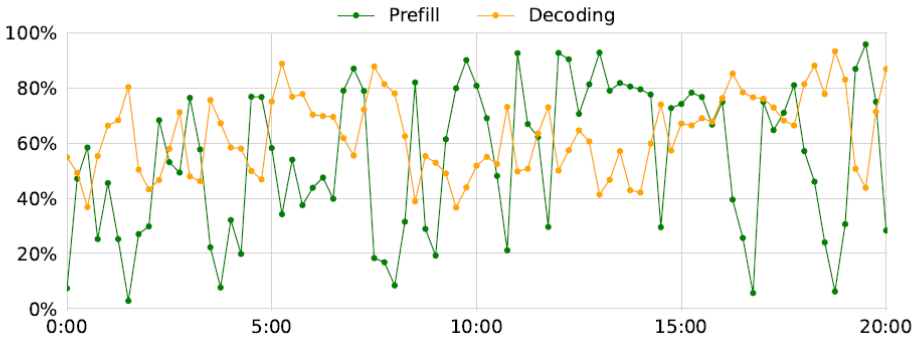
> Estimate requests still decoding when THIS request finishes prefill

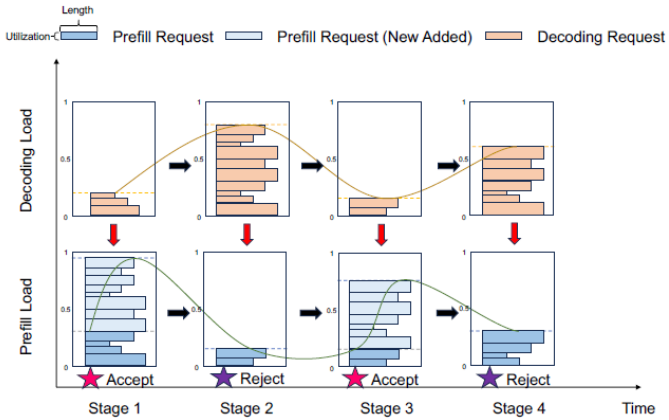
> Calculate predicted TBT ratio at that future time point

> Accept if predicted future load is within TBT SLO threshold

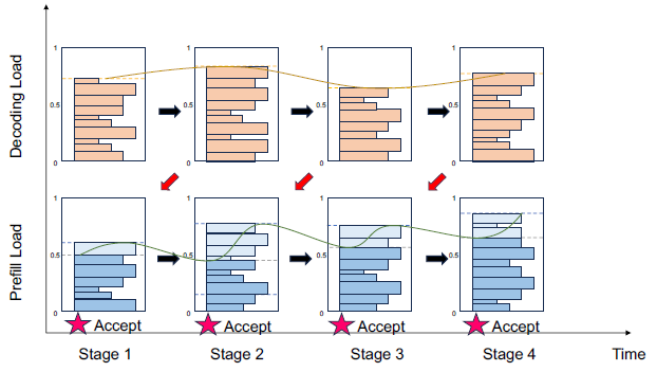
> Reject NOW if future decoding will exceed capacity

Result: smooth stable load. No oscillation. Higher cluster utilisation.





(a) Early Rejection.



(b) Early Rejection Based on Prediction.

Key Hardware: VRAM, CPU DRAM & GPUDirect RDMA

GPU VRAM

- Video RAM -- physically on the GPU chip
- Bandwidth: ~2,000 GB/s
- Capacity: 40-80 GB
- Cost per GB: ~$25
- Only GPU can access directly
- All matrix multiplications MUST happen here
- KV cache for active requests must be here
- Limited size is the primary bottleneck

CPU DRAM

- Main system RAM on the motherboard
- Bandwidth: ~150 GB/s
- Capacity: 512 GB - 2 TB per server
- Cost per GB: ~$5
- CPU accesses directly; GPU via PCIe
- Mooncake uses DRAM as KV cache pool
- Large enough for cross-request prefix cache
- Cheap enough to scale across the cluster

GPUDirect RDMA

- Remote Direct Memory Access
- Bandwidth: up to 800 Gbps between nodes
- Latency: microseconds
- Normal path: GPU -> CPU copy -> NIC -> network
- RDMA path: GPU -> NIC directly (CPU bypassed)
- CPU only issues 'start transfer' command
- Data moved by NIC + GPU DMA engine directly
- Achieves full 800 Gbps without CPU bottleneck

Evaluation Results

20%
Throughput vs vLLM-4M
(ArXiv dataset)

40%
Throughput vs vLLM-4M
(L-Eval, 80% cache)

525%
Peak gain
128k context

75%
More requests
Kimi production

Key Findings

> Gains grow with context length: 16k=50%, 32k=100%, 64k=200%, 128k=525% -- because long prefills disrupt decoding most in coupled systems like vLLM

> L-Eval (80%+ cache reuse) shows larger gains than ArXiv (~0% cache) -- prefix caching dramatically cuts prefill time for repeated contexts

> Mooncake [3 prefill + 1 decode nodes] outperforms vLLM [4 coupled nodes] on both TTFT and TBT simultaneously across all workloads

> Hardware: 8x NVIDIA A800-SXM4-80GB per node, 800 Gbps RDMA. Experiments use LLaMA2-70B to protect Kimi proprietary weights.

How All Components Work Together: Request Lifecycle

1	Request arrives at Conductor
2	Hash prompt tokens -- find prefix match in distributed DRAM pool
3	Select prefill node: cache-aware + load-balanced + TTFT estimated
4	Load cached KV (DRAM to VRAM), run incremental prefill, store new KV
5	Layer-wise async stream KV to decoding node via GPUDirect RDMA
6	Decoding node: async load KV (DRAM to VRAM), join continuous batch
7	Generate output tokens one by one, stream response to user

Overload path: Conductor predicts future decoding load and rejects BEFORE prefill if decoding will be full , HTTP 429, zero compute wasted.

Key Takeaways

[1] **KV Cache is the Central Problem** -- In LLM serving at scale, KV cache management determines throughput, latency, and cost.

[2] **Disaggregation Eliminates Interference** -- Separating prefill and decoding onto different machines removes the fundamental TTFT vs TBT tension.

[3] **CPU DRAM is an Underused Asset** -- Aggregating cluster-wide CPU DRAM into a shared KV pool provides far more capacity than GPU VRAM alone.

[4] **Hash-based Reuse Across Requests** -- Prefix-chained hashes enable cross-request KV sharing with no GPU compute -- just integer comparison.

[5] **Overload is the Real Operating Condition** -- Production systems must reject intelligently. Prediction-based early rejection prevents oscillation and waste.