

Mooncake: The KVCache Is the Thing You Schedule

Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving

Qin, Li, He, Zhang, Wu, Zheng, Xu · Moonshot AI + Tsinghua · FAST '25 · arXiv:2407.00079 · Day 4

Why this one matters more than the other three. Days 1–3 were research systems measured on replayed traces. Mooncake is **the platform that actually runs Kimi**, a live commercial product, under permanent overload. It is the only paper in the series that answers the question a Google interviewer will eventually ask: "*what happens when you can't serve everyone?*"

Words you need first

Prefill / Decode / KVCache / TTFT / TBT / SLO

Same as Day 3. Quick refresh: **prefill** reads your whole prompt at once and emits the first word. **Decode** writes the rest one word at a time. **KVCache** is the scratch notes about every token seen so far. **TTFT** = wait until the first word. **TBT** = gap between words. **SLO** = the speed promise. (This paper says TBT; DistServe said TPOT. Same thing.)

Prefix cache

If two requests start with the same text, their KVCache for that shared beginning is **byte-identical**. So compute it once, store it, and let the second request skip straight past it. Everyone reusing the same system prompt, or asking follow-up questions about the same document, gets this for free — if your system is built to notice.

Prefix hash

How you notice. Chop the prompt into blocks of 512 tokens. Hash each block **together with the hash of everything before it**. Two blocks match only if the entire preceding text also matched — which is exactly the condition for the cached KV to be valid. Identical hash = safe to reuse.

VRAM vs DRAM vs SSD

Three tiers of storage, fastest and scarcest first. **VRAM** = memory on the GPU card (80 GB, precious). **DRAM** = ordinary CPU RAM in the same server (hundreds of GB, mostly idle in a GPU cluster). **SSD** = disk (terabytes, slow but nearly free). Mooncake's trick is to use the second and third tiers, which everyone else was leaving empty.

RDMA / GPUDirect

RDMA (Remote Direct Memory Access) lets one machine read another machine's memory *without involving either CPU* — like a direct pipe between two buildings. **GPUDirect** extends that straight into GPU memory, skipping the CPU entirely. This is what makes a cluster-wide cache viable instead of a fantasy.

MFU — Model FLOPs Utilization

What fraction of the GPU's theoretical math capacity you're actually using. Low MFU = you paid for a Ferrari and drove it in traffic. Decode has terrible MFU; prefill has good MFU.

Tensor parallelism (TP) / Sequence parallelism (SP) / Pipeline parallelism

TP: split each math operation across GPUs — fast, but they must talk constantly. **SP**: split the *input sequence* across GPUs (one GPU takes tokens 1–8000, another 8001–16000) and pass partial attention results around; used for very long prompts. **Pipeline**: assembly line — GPU 1 does the first layers, GPU 2 the next, and they only talk at the handoff.

Goodput — Mooncake's stricter version

DistServe counted a request as "good" if it met its SLO. Mooncake is harsher: **only requests that fully complete count**. If you do the expensive prefill and then have to drop the request, every token you computed is thrown away and counted as zero. This single definitional change is what forces the whole second half of the paper.

Admission control / Early rejection

Deciding at the front door not to accept a request at all. Sounds like giving up; it's actually the highest-leverage decision an overloaded system makes. A request you never start costs nothing. A request you start and abandon costs everything.

HTTP 429

The standard web response code for "too many requests, try again later." Mooncake returns this when it can predict up front that it cannot meet your SLO.

Conductor / Messenger

Mooncake's two named components. **Conductor** = the global brain that decides which prefill machine and which decode machine handle each request. **Messenger** = a process on every node that moves KVCache blocks over RDMA.

Problem

Every throughput trick you have breaks an SLO

There are only two ways to raise throughput, and each one damages a different latency metric:

Throughput lever	What it buys	What it breaks
Reuse the KVCache more (prefix caching)	Skip computation entirely	Fetching cache from a remote machine takes time → TTFT goes up
Put more tokens in each batch	Better MFU	Bigger batch = slower iteration → TBT goes up

So you can't just turn both dials up. Every decision is a three-way trade between compute saved, TTFT, and TBT — and that is precisely what Conductor is built to arbitrate.

What Kimi's real traffic actually looks like

They open-sourced a 1-hour trace of 23,608 real requests (timestamps, lengths, and block hashes — no user content). The numbers are worth memorizing because they're the only public production data of this kind:

- **Average input: 7,590 tokens. Average output: 182 tokens.** Input dwarfs output by roughly 40x. This is a *reading* workload, not a writing one — so prefill cost, not decode cost, dominates.
- **Cache hit rate: 30%** with a 1,000-block cache, rising to **50%** at 50,000 blocks — then flat. Bigger caches stop helping.
- **LRU beat LFU** and a length-aware policy on this trace, because reuse clusters in time (a user asks follow-ups about the same document).
- **Brutal skew: over 50% of cache blocks are never reused at all**, while some are read tens of thousands of times. A shared system prompt is touched by everyone; your uploaded PDF is touched by you.
- **Honest ceiling:** even with infinite storage and infinite time budget, only ~50% of KVCache is reusable on this workload. For their chat-to-paper service it reaches 90%. **Reuse is entirely application-dependent** — never quote a single number as universal.

That skew is the whole reason cache scheduling is hard. If every block were equally popular, you'd hash-partition them across machines and be done. Instead, a handful of blocks are wanted by nearly every request — so the machine holding them becomes a traffic jam, no matter how much cache it has.

The problem nobody else studied: permanent overload

Every paper so far assumed every request gets served, and asked how to serve them efficiently. Kimi cannot make that assumption. User growth outruns GPU supply, and you can't buy your way out during a shortage. So the real question becomes **which requests to refuse**.

Plain version: A restaurant at full capacity has two options. Option A: seat everyone, cook every dish halfway, and throw out the food when tables need turning. Option B: tell people at the door "we can't seat you tonight." Option B looks worse and is dramatically better — nothing is wasted. Mooncake's whole second half is about becoming very good at saying no, *early*.

And there's a specific trap. In a disaggregated system, prefill and decode are scheduled at **different moments**. A request can clear the prefill check, consume seconds of expensive prefill compute, and then get rejected by the decode machine because load changed in the meantime. That's the worst outcome available: full cost, zero value.

Intuition

Reframing #1 — schedule around the cache, not around the GPUs. Everyone else asks "which GPU is least busy?" Mooncake asks "where does the data this request needs already live?" — and only then, "can that machine take it?" Once the KVCache becomes the primary object, prefix caching, load balancing, replication, and admission control all turn out to be one problem instead of four.

Reframing #2 — your cluster is already full of unused storage. A GPU node has 8 expensive GPUs and also hundreds of gigabytes of CPU DRAM, terabytes of SSD, and RDMA cards, all mostly idle during inference. Pool them into a cluster-wide KVCache. You get an enormous near-GPU cache **at zero additional hardware cost**.

Plain version: Your GPUs are chefs. Their VRAM is the countertop — tiny, and you need it for actual cooking. But there's a whole walk-in fridge (CPU DRAM) and a basement pantry (SSD) sitting empty three metres away. Mooncake stocks the fridge with prepped ingredients (cached KV) so the chefs almost never start from raw.

Reframing #3 — under overload, the most valuable action is refusing work early. Because only complete requests count, a rejection at the door costs nothing while a rejection after prefill costs everything. Admission control stops being a safety valve and becomes a first-class optimization.

Solution

1. The architecture

- **Prefill pool** — GPU nodes that only do prefill, organized for long contexts.
- **Decode pool** — GPU nodes that only do decode, running continuous batching.
- **Disaggregated KVCache pool** — CPU DRAM and SSD across the whole cluster, holding paged KVCache blocks. Evicted by LRU/LFU.
- **Conductor** — global scheduler. Picks the (prefill, decode) pair, and replicates or swaps cache blocks.
- **Messenger** — per-node process doing GPUDirect-RDMA block transfers.

The cache is **content-addressed**: each block gets a hash of its own tokens concatenated with the previous block's hash. Identical hash means identical text *and* identical history, so reuse is always safe, and duplicates deduplicate themselves.

2. A request's four steps

1. **KVCache reuse.** Load whatever prefix cache already exists from remote CPU memory into the chosen prefill node's GPU. Skipped if there's no match.
2. **Incremental prefill.** Compute only the uncached remainder. If that remainder exceeds `prefill_chunk` (typically >1000 tokens), split it into chunks and pipeline them.
3. **KVCache transfer.** Stream the new KV to the decode node **layer by layer, overlapped with computation** — not as one blocking copy at the end.
4. **Decode.** Join the continuous batch. The decode node **re-checks** the TBT SLO locally, because load may have shifted since Conductor decided. If it fails now, the request is dropped and the prefill work is wasted — which is exactly what early rejection exists to prevent.

3. Chunked pipeline parallelism (CPP) for long prompts

A 128K-token prompt needs more than one 8-GPU node. Three options, and they rejected two:

Option	Why not / why yes
TP across nodes	No. Two RDMA all-reduces <i>per layer</i> across the network. MFU collapses.
Sequence parallelism	No. Better than TP, but still cross-node communication at least once per layer. Still worse MFU than single-node TP. Also needs elastic resizing of the SP group as prompt lengths vary, which complicates Conductor. And it competes for the same network bandwidth the KVCache transfers need.
CPP (their choice)	Yes. Group X nodes into a pipeline. Split the prompt into chunks; different chunks run on different nodes simultaneously. Communication happens only at stage boundaries and overlaps with compute. Handles short and long prompts with the same static setup — no dynamic repartitioning. They believe this is the first use of pipelining for <i>inference</i> prefill.

4. Layer-wise prefill — and why it removes VRAM from the equation

Prefill runs layer by layer and is compute-bound, so there's spare memory bandwidth. Mooncake overlaps cache movement with computation: before layer *N*'s attention, wait for layer *N*'s KV load and kick off layer *N+1*'s; after attention, launch layer *N*'s async store.

Net effect: prefill wall-clock time $\approx \max(\text{KV load time}, \text{normal prefill time})$ instead of the sum.

The consequence is the interesting part. Since KV never piles up in VRAM, **VRAM stops being a scheduling constraint** — as long as a single request fits. Prefill scheduling then only has to consider cache locations and DRAM. Removing a constraint from the scheduler is often worth more than the speedup that produced it. (Their forward-looking use for that freed VRAM: run cheap 24-hour-turnaround batch-API requests on prefill nodes, since those have no TBT requirement at all.)

5. Why they keep disaggregation even though chunked prefill exists

This is the paper explicitly refereeing the Day 2 vs Day 3 argument, from production experience. They **concede** chunked prefill's advantages: uniform nodes make scheduling easier, and inlining prefill raises the decode batch's MFU. They keep disaggregation anyway, for two reasons:

1. Prefill nodes need **different cross-node parallelism** (CPP) to handle long contexts. You can't have that if all nodes are identical.
2. Layer-wise prefill's **VRAM saving** only exists on dedicated prefill nodes. Cost of holding a request's KV is `size × time`; inlining a chunked prefill into a decode batch *stretches the time*, so it costs more VRAM.

But the actual answer is a hybrid, and this is the line to remember: Mooncake **does** inline a prefill into the decode batch — when it can be done *without chunking* and *without breaking the TBT SLO*. Short requests get the chunked-prefill treatment; long ones go to the dedicated pipelined prefill pool. Not either/or. Both, chosen per request.

6. KVCache-centric scheduling (Algorithm 1)

For every incoming request, hash its blocks, then for each prefill instance estimate:

```
TTFT_estimate = T_queue + T_prefill (+ T_transfer, if pulling cache from elsewhere)
```

- **T_prefill** comes from an offline-fitted model of (prompt length, prefix hit length). Accurate because transformer compute is highly regular.
- **T_queue** = sum of prefill times of everything already queued on that instance.
- **T_transfer** is the hard one — it depends on live network congestion, not just data size.

Pick the instance with the lowest estimated TTFT. **If even the best option misses the SLO, return HTTP 429 immediately** rather than accepting a request you know will fail.

Hot-spot replication that nobody planned

The elegant bit. They **refuse** to predict future block popularity — workloads change too fast at a fast-growing company. Instead, a threshold: if an instance's local prefix match is much worse than the global best, and fetching remotely would be faster than recomputing, then fetch it — and the fetching instance **keeps a copy**.

So every time load pushes a request away from the cache-richest node, the hot block gets replicated one more time. **Popular blocks spread themselves across the cluster automatically; unpopular ones never do.** No prediction, no global usage tracking — a local rule producing the right global behaviour. That's the kind of design an interviewer remembers.

7. Overload scheduling — and the oscillation bug

Measuring load correctly

In a coupled system you can't predict TTFT or TBT (interference makes them unpredictable), so people fall back to counting requests. Disaggregation removes the interference, so Mooncake can use **predicted SLO satisfaction as the load metric directly** — compare predicted max TTFT against the TTFT limit, predicted TBT against the TBT limit. *Disaggregation didn't just improve performance; it made the system observable.*

Early rejection

Check **both** pools' load before starting prefill, not just the prefill pool's. Rejects doomed requests before spending anything.

The surprise: early rejection makes the cluster oscillate

On a 20-machine cluster they observed prefill and decode loads swinging in **anti-phase** — one pool saturated while the other idled, alternating. Four-stage cycle:

1. Both loads low → accept lots of requests → prefill saturates.
2. Those requests land on decode → decode saturates → start rejecting → **prefill goes idle**.
3. No new arrivals reach decode → decode drains → load drops.
4. Accept lots again → back to stage 1.

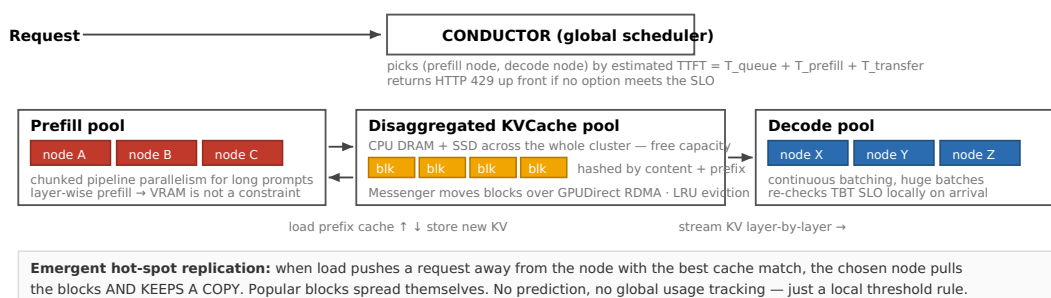
Root cause: control-loop lag. You're deciding based on decode load *right now*, but the consequence of that decision arrives one full prefill-duration later. Classic delayed feedback, and it produces classic oscillation. Worse with fewer prefill machines and longer prefills.

The fix: predict the load you're about to cause

Don't reject based on current decode load — reject based on **predicted decode load after this prefill finishes**. Two ways:

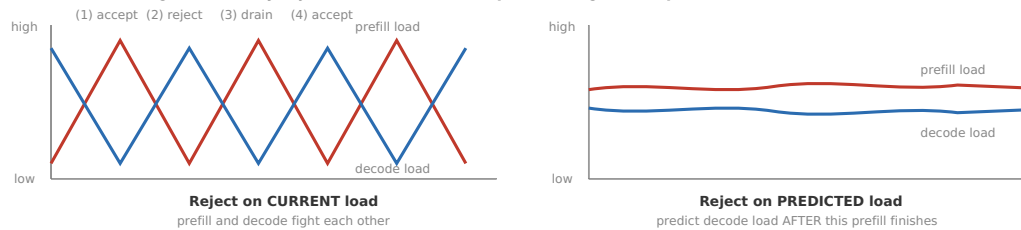
- **Request level** — predict each request's output length. Would be ideal, but it's expensive or inaccurate, and hardest exactly when you need it most (under overload). Left as future work.
- **System level** (what they ship) — assume every request's decode takes a uniform time **t_d**. Roll the simulation forward to time *t*: add requests whose prefill will have finished, remove requests whose decode will have exceeded **t_d**, then compute the average TBT ratio across decode instances. Coarse, continuous, and good enough — precision isn't needed, direction is.

Solution Diagram



Mooncake architecture. The cache pool in the middle is the part no earlier paper had.

The oscillation bug: naive early rejection makes the two pools swing in anti-phase



Root cause: you decide using decode load NOW, but your decision lands one prefill-duration later. Delayed feedback → oscillation.

A control-theory failure hiding inside a scheduling policy.

Code

```
# ===== 1. PREFIX HASHING – how blocks become reusable and self-deduplicating =====
# A block is only safe to reuse if ALL PRECEDING text also matched.
# Chaining the previous hash into this one encodes exactly that condition.

BLOCK = 512 # tokens per block, as used in the released trace

def prefix_hashes(tokens):
    keys, prev = [], ""
    for i in range(0, len(tokens), BLOCK):
        blk = tokens[i : i + BLOCK]
        prev = sha256(prev + str(blk)).hexdigest() # chain with the previous
        keys.append(prev)
    return keys

# Two requests sharing the first 12 keys share 12*512 = 6,144 tokens of KVCache.

# ===== 2. KVCACHE-CENTRIC SCHEDULING (Algorithm 1) =====
def schedule(request, prefill_pool, decode_pool, threshold):
    keys = prefix_hashes(request.tokens)
    best_len, best_holder = find_best_prefix_match(prefill_pool, keys) # global best

    best_ttft, chosen = INF, None
    for inst in prefill_pool:
        local_len = inst.match(keys) # what THIS node already has
        t_queue = estimate_queue_time(inst) # sum of queued prefill times

        if best_len / max(local_len, 1) < threshold:
            # Local cache is nearly as good as the best. Just compute here.
            ttft = t_queue + estimate_prefill(len(request.tokens), local_len)
        else:
            # Local cache is much worse. Consider PULLING the remote blocks.
            t_xfer = estimate_transfer(best_holder, inst, best_len - local_len)
            ttft = t_queue + t_xfer + estimate_prefill(len(request.tokens), best_len)

        if ttft < best_ttft:
            best_ttft, chosen = ttft, inst

    d, tbt = select_decoding_instance(decode_pool) # load-balanced

    # ADMISSION CONTROL: refuse up front rather than fail expensively later.
    if best_ttft > TTFT_SL0 or tbt > TBT_SL0:
        return HTTP_429

    # HOT-SPOT MIGRATION, as a side effect: the puller keeps its copy.
    if best_len / max(chosen.prefix_len, 1) > threshold:
        transfer_kvcache(best_holder, chosen) # popular blocks self-replicate

    return chosen, d

# ===== 3. EARLY REJECTION BASED ON PREDICTION – the oscillation fix =====
# WRONG: reject on decode load NOW. Your decision lands one prefill later.
def admit_naive(req, decode_pool):
    return current_load(decode_pool) < 1.0 # causes anti-phase oscillation

# RIGHT: roll the world forward to when this prefill will actually finish.
def admit_predictive(req, prefill_pool, decode_pool, t_d):
    t = now() + estimate_prefill_completion(req, prefill_pool)

    projected = list(decode_pool.active)
    projected += [r for r in prefill_pool.in_flight if r.finishes_by(t)] # arrivals
    projected = [r for r in projected if r.decode_elapsed_at(t) < t_d] # departures

    # t_d = assumed uniform decode duration. Crude, but it only needs
    # to get the DIRECTION right, not the exact value.
    return predicted_tbt_ratio(projected) < 1.0
```

Results

Experiment (vs vLLM, same total node count)	Result
ArXiv Summarization (~0% cache reuse), 3P+1D vs vLLM-[4M]	+20% throughput
L-Eval (>80% cache reuse), 3P+1D vs vLLM-[4M]	+40% throughput
Simulated long context, 16K-128K prompts	+50% to +525%
Real Kimi trace, 10P+10D vs vLLM-[20M]	+75% requests served

The real-workload result is the one to quote, and the detail matters. TTFT distributions were **nearly identical** — both systems passed the TTFT SLO on ~100% of requests. The entire difference was TBT: Mooncake passed on ~100%, **vLLM on only 57%**. If you had monitored TTFT alone, you would have concluded the two systems were equivalent. *The metric you don't graph is the one that's broken.*

Overload ablation (8P+8D, 23,000 real requests replayed at 2× speed):

Rejection strategy	Requests rejected
Baseline (reject at each stage independently)	4,183
Early rejection (check both pools up front)	3,771
Early rejection based on prediction	3,589

Roughly 14% fewer rejections from scheduling changes alone — no extra hardware. Modest-looking, but it's free capacity on a fleet you can't grow.

Scheduling ablation (8P+8D, 23,000 requests): random < load-balancing < cache-aware < full KVCache-centric, on both average TTFT and TTFT SLO attainment. Cache-awareness beats load-awareness; combining them beats either.

A negative result they report honestly: the 2P+2D configuration had *worse* TTFT than 3P+1D despite better TBT — the prefill:decode ratio was wrong for that workload. The ratio is workload-specific, currently preset, and getting it wrong costs you real performance.

Likely Interview Questions + Answers

Q1. Why does chaining the previous block's hash into the current one matter? Why not hash each block independently?

Because KVCache validity depends on **everything that came before**. The KV vectors for tokens 513-1024 were computed while attending to tokens 1-512; if a different request has the same tokens 513-1024 but different tokens 1-512, its KV would be **different**, and reusing the cached version would be silently wrong. Chaining makes a hash match imply "same content *and* same history," which is exactly the reuse precondition. It also gives free deduplication, since identical prefixes collapse to one stored copy. Generalizes to any content-addressed cache where entries depend on context.

Q2. How do you handle cache hot spots without predicting which blocks will be popular?

You don't predict — you let replication emerge from a local rule. Mooncake's threshold says: if this node's local prefix match is much worse than the global best, and pulling the remote blocks beats recomputing them, pull them *and keep the copy*. Every time load diverts a request away from the cache-richest node, the hot block gets one more replica. Popular blocks spread; unpopular ones never do. No global usage tracking, no forecasting model. They chose this deliberately because at a fast-growing company the workload distribution shifts faster than any predictor can learn it. **The reusable lesson: prefer a local rule with the right emergent behaviour over a global optimizer that needs a forecast you can't produce.**

Q3. What is "early rejection" and why does the naive version make things worse?

Early rejection means checking *both* the prefill and decode pools' load before starting prefill, so you never spend expensive prefill compute on a request that decode will drop. The naive version rejects based on decode load **right now** — but the request you're admitting won't reach decode until one prefill-duration later. That lag is delayed feedback, and delayed feedback in a control loop produces oscillation: prefill saturates → those requests hit decode → decode saturates → reject everything → prefill goes idle → decode drains → accept a flood → repeat. The two pools end up in anti-phase, each idle while the other is pinned. The fix is to predict decode load *at the moment this prefill will finish*, not now. **This is a control-theory answer to a scheduling question and it lands well in interviews** — the same failure shows up in autoscalers that react to current CPU while instances take minutes to boot.

Q4. Why is Mooncake's definition of goodput stricter than DistServe's, and what does that change?

DistServe counts a request as good if it met its SLO. Mooncake counts it only if it **fully completed** — a request abandoned mid-flight contributes zero, and every token computed for it is waste. That's the right definition for a commercial service, where a half-generated answer has no value and generates no revenue. The consequence is that **admission control becomes a primary optimization rather than a safety valve**: since partial work is worthless, refusing early is strictly better than failing late, and the whole overload-scheduling section exists because of this one definitional choice. Watch for the general pattern — changing the objective function is often what unlocks a design, not changing the mechanism.

Q5. Chunked prefill or disaggregation? Mooncake runs in production — what did they actually pick?

Both, chosen per request. They keep disaggregation for two production reasons: prefill nodes need different cross-node parallelism (chunked pipeline parallelism) to handle 128K-token prompts, which is impossible if all nodes are identical; and layer-wise prefill's VRAM saving only exists on dedicated prefill nodes, because inlining a chunked prefill into a decode batch stretches how long the KV sits in VRAM (cost is size × time). But they **do** inline a prefill into the decode batch when it fits without chunking and without breaking the TBT SLO. So: short requests get the chunked-prefill treatment, long ones go to the pipelined prefill pool. If asked to pick a side, the strongest answer is that production converged on a hybrid, and to give the specific condition under which each path is taken.

Q6. Why does the cache live in CPU DRAM and SSD instead of GPU memory?

Capacity and cost. GPU VRAM is ~80 GB per card and every gigabyte spent on cache is a gigabyte not available for a larger decode batch. Meanwhile a GPU node has hundreds of gigabytes of CPU DRAM and terabytes of SSD sitting almost completely idle during inference. Pooling those across the cluster gives an enormous cache for **zero additional hardware**. GPUDirect RDMA is what makes it practical — data moves between machines without either CPU touching it. Layer-wise prefill then hides the load latency behind computation, so the tier below VRAM costs almost nothing in wall-clock terms. **The general move: find the resource your workload already pays for but doesn't use.**

Q7. Why chunked pipeline parallelism instead of sequence parallelism for long prompts?

Both split a long prompt across nodes. Sequence parallelism communicates **at least once per layer** across the network — better than cross-node tensor parallelism (two all-reduces per layer) but still worse MFU than staying inside one node, and it competes for the very bandwidth the KVCache transfers need. It also demands dynamically resizing the SP group as prompt lengths vary, which complicates the global scheduler. CPP communicates **only at pipeline stage boundaries**, and that communication overlaps with computation. It also handles short and long prompts with one static configuration, so no repartitioning. The trade-off you should name: CPP doesn't reduce the latency of a single short request the way SP can, but it wins on throughput, on network contention, and — most importantly in production — on operational simplicity.

Q8. What's the realistic prefix cache hit rate, and how much should I count on it?

On Kimi's real trace: 30% with a small cache, rising to 50% with a large one, then **flat** — more capacity stops helping. And ~50% is the theoretical ceiling for that workload even with infinite storage and infinite time. But their chat-to-paper service reaches 90%. **Reuse is a property of the application, not of the system.** Shared system prompts, multi-turn conversations, and repeated-document workloads cache well; one-shot diverse queries don't. Two things to say in an interview: first, measure your own reuse before designing around it; second, note the extreme skew — over half of all blocks are never reused while a few are read tens of thousands of times — which means your cache design problem is really a hot-spot problem, not a capacity problem.

Q9. Design: you're serving an LLM under permanent overload and can't get more GPUs. What do you do?

Reframe from "serve everything" to "choose what to serve." (1) **Define goodput as fully-completed requests**, so partial work counts as zero — this makes the rest of the reasoning fall out. (2) **Admission control at the door**: estimate TTFT and TBT before accepting; return 429 immediately if you can't meet them. Cheap to compute, and it's free capacity. (3) **Predict, don't react** — reject based on load at the time your decision lands, not current load, or you'll oscillate. (4) **Cut work, don't just schedule it**: prefix caching removes computation entirely, which beats any scheduling improvement; measure your reuse rate first. (5) **Use idle tiers** — CPU DRAM and SSD for cache, since you already own them. (6) **Add priorities**, so shedding hits the cheapest traffic first (Mooncake names this as future work). (7) **Monitor both TTFT and TBT** — in their real-workload test the two systems had identical TTFT and the entire difference was in TBT. The framing that earns the point: **under overload, deciding what NOT to do is the primary optimization.**

Q10. What are you most worried about operating this?

Three things. **Prefill:decode ratio** is preset and workload-specific — their own 2P+2D config had worse TTFT than 3P+1D. Get it wrong and you've stranded capacity in the wrong pool, with no dynamic conversion available. **Cache hot spots** self-replicate only reactively, so a sudden viral prompt causes congestion before replication catches up. **Fault tolerance** is entirely undiscussed — a disaggregated architecture with a central Conductor and cross-pool dependencies has failure modes a replicated colocated fleet doesn't. I'd also flag something the paper doesn't discuss at all: a **cluster-wide prefix cache shared across users** is a potential information-leak surface, since cache-hit timing can reveal whether someone else has submitted the same text. That's my own observation rather than theirs, but it's exactly the kind of thing a senior interviewer wants to hear raised unprompted.

What's not covered

Named as future work by the authors

- **Request-level output-length prediction.** Would make load prediction far better; too costly or inaccurate today, and worst exactly under overload.
- **Dynamic prefill:decode rebalancing.** Ratios are currently preset. No live conversion of a node from one pool to the other.
- **Request priorities and per-tenant SLOs.** Today all requests are equal, so shedding can't protect premium traffic.
- **Heterogeneous accelerators.** Their strongest forward-looking argument: judged on bandwidth-per-dollar or bandwidth-per-watt, plain GDDR/LPDDR memory can beat flagship GPUs by an order of magnitude — ideal for the memory-bound decode phase. Also process-in-memory and hybrid bonding.
- **Separating the attention operator from the linear operators.** Attention's arithmetic intensity depends only on heads + KV-heads and *cannot* be improved by bigger batches, so it's permanently more memory-bound than everything else — a natural third pool. Simulated only. (They note DeepSeek-v2's MLA attacks the same problem from the model side instead.)
- **Smarter cache eviction** for partial hits and expiry.

Evaluation limits

- **A dummy model.** All results use a placeholder with LLaMA2-70B's architecture, and the trace has no real text — only lengths and remapped hashes. Fine for systems claims, useless for anything about output quality.
- **vLLM is the only baseline.** No comparison against DistServe, Splitwise, or Sarathi-Serve, despite all being discussed. So the papers in this series still have never been benchmarked against each other.
- **Excellent networking assumed.** Testbed: A800 GPUs with NVLink inside nodes and **800 Gbps RDMA** between them. The design leans on this; behaviour on commodity networking is untested.
- **The balancing threshold is tuned by hand.** They flag adaptive tuning as future work.
- **One hour of traffic from one product.** Long-context Chinese-language assistant usage. Generalize carefully.

Not discussed at all

- **Fault tolerance and failure recovery** — a notable gap for a production paper, and disaggregation makes it harder (as DistServe admitted).
- **Security and isolation of a shared cross-user cache.** Timing side channels on cache hits are a real concern that the paper never raises.
- **Cost accounting** — no dollars-per-request numbers, despite revenue being the stated motivation.
- Speculative decoding, quantization, MoE routing.

Where the series stands after four days. You can now argue the full stack: **memory** (PagedAttention) → **scheduling** (Sarathi-Serve) → **placement** (DistServe) → **cache + overload** (Mooncake). That's a complete answer to "design an LLM serving system," including the part most candidates miss — what happens when demand exceeds supply.

Natural next steps: Splitwise (disaggregation with *different GPU types* per phase — the heterogeneity gap all three papers point at) → Lumnix (live request migration between instances, which fixes Mooncake's static P:D ratio) → then step outside serving entirely, since your interview loop will also want training-side systems (ZeRO, Megatron-LM) and classical distributed systems.