Mooncake:
A KVCache-centric Disaggregated Architecture for LLM Serving

CS2650

Presented by

Raima Afra
Tom Tan
Aadity Sharma
Andrew Zhao

MOTIVATION

LLM serving at scale is a resource scheduling problem

Throughput vs Latency Trade-off in LLM Inference

- Higher throughput
- More KVCache needed

- Smaller batches
- Less KVCache pressure

Business Goals

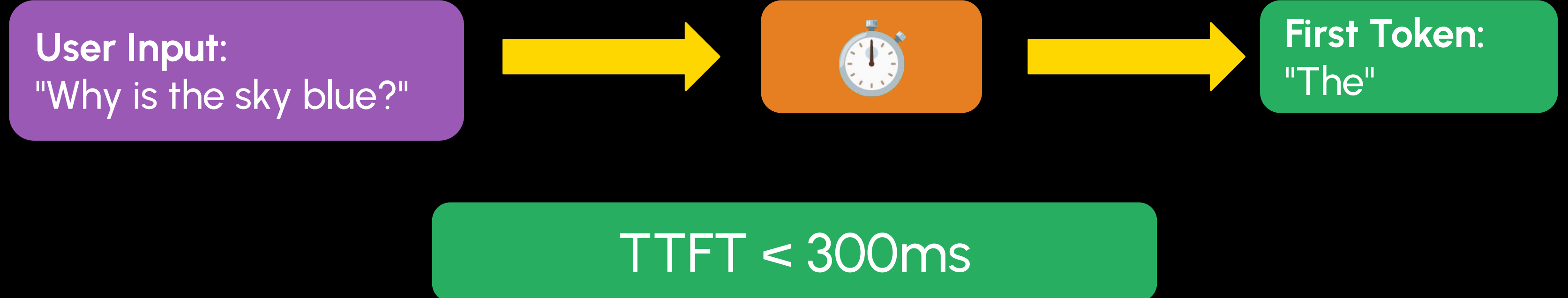
Maximize Throughput

More users =
More revenue

Meet Service Level Objectives

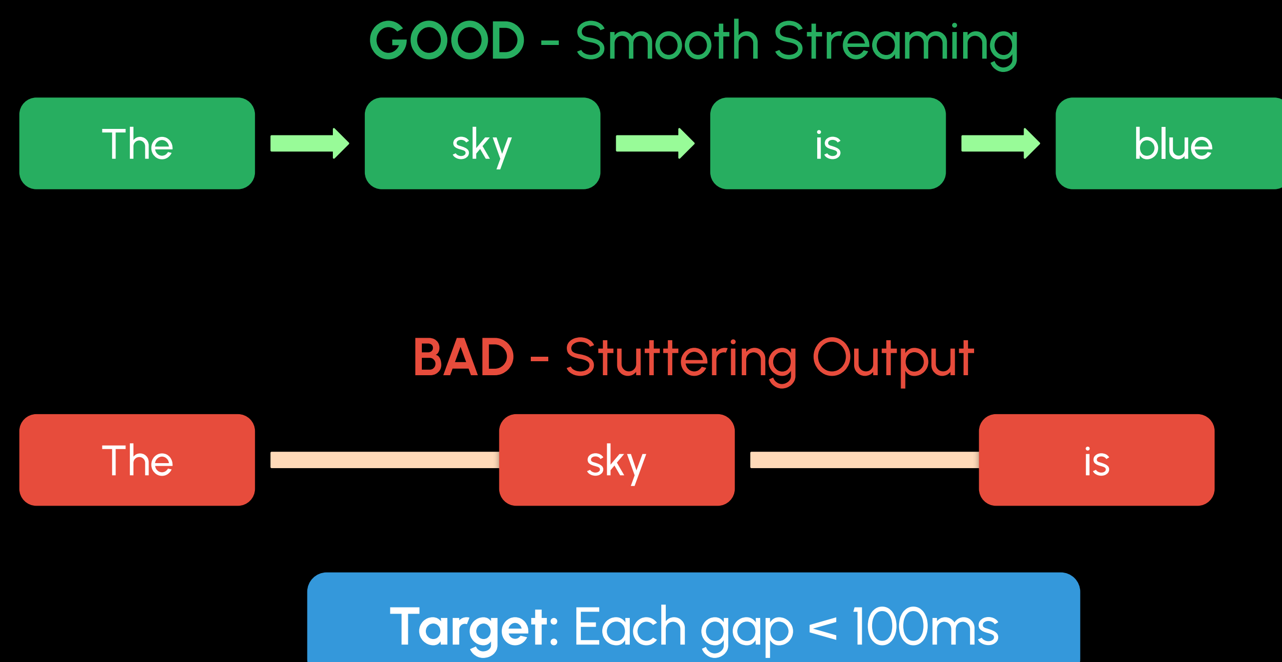
Fast responses =
Happy users

Key Metric #1: TTFT (Time to First Token)



Users perceive this delay immediately

Key Metric #2: TBT (Time Between Tokens)



How LLMs Generate Responses: Two Stages

PREFILL STAGE

Input: "Why is the sky blue?"
(all tokens at once)

Process: Parallel ⇉⇉⇉

Output: First token "The"
+ KVCache stored

Characteristics:
· Memory-bound
· High GPU usage
· One-time cost

DECODING STAGE

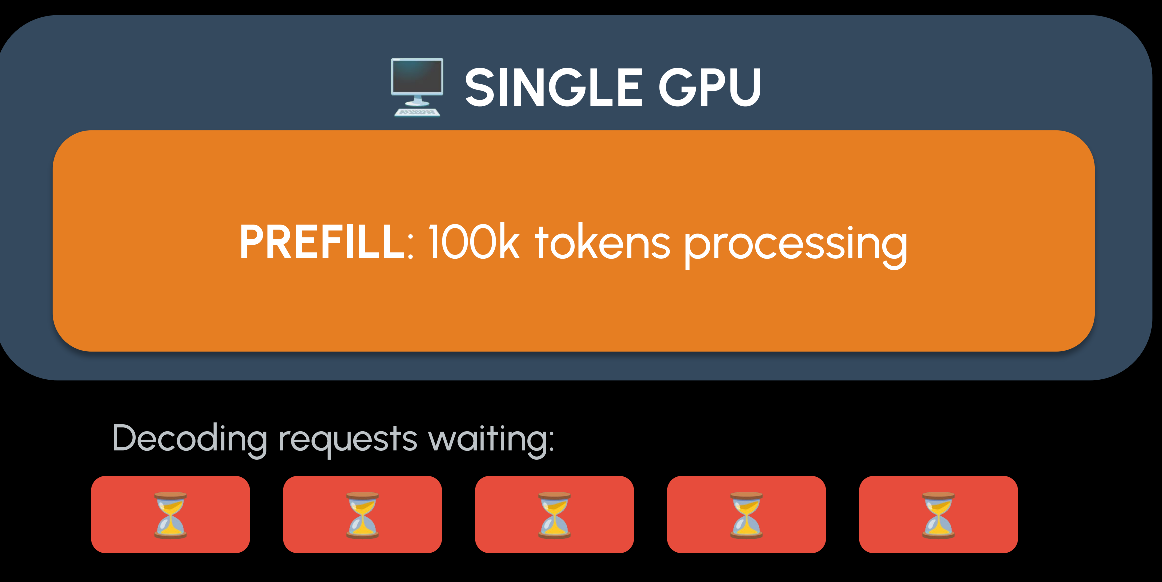
Input: Previous context
+ last token

Process: Sequential ⛓

"sky" "is" "blue"

Characteristics:
· Memory-bound
· Sequential
· Repeats many times

The vLLM Approach: Prefill + Decoding on SAME GPU



Mooncake's Core Insight

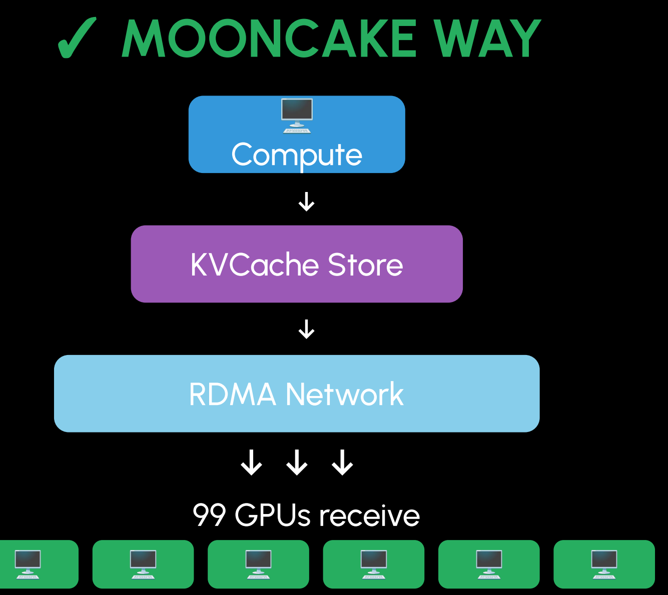
Same 10k token system prompt used 100 times

❌ TRADITIONAL WAY

All 100 GPUs computing

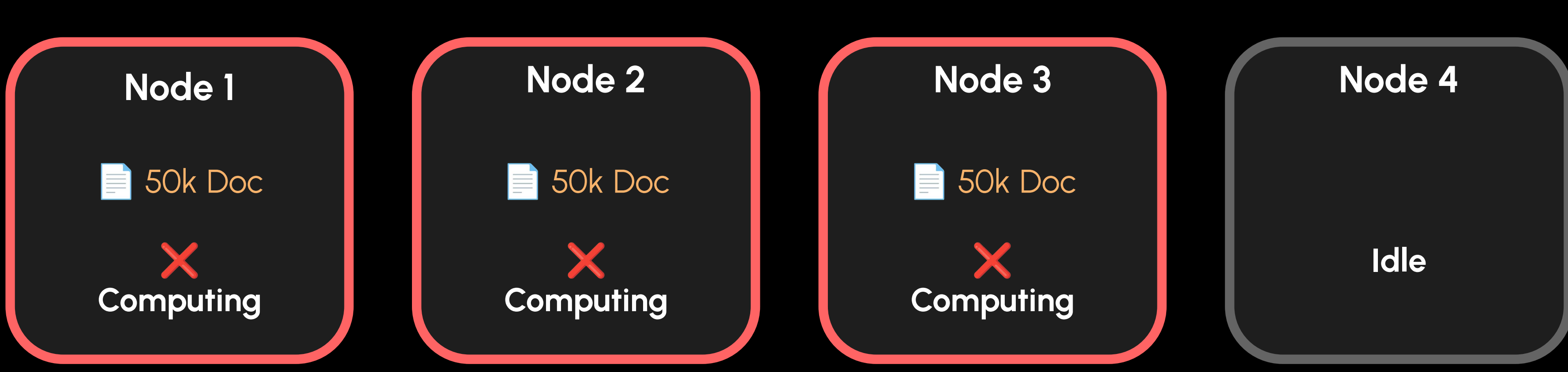
Total Cost:
!! 100 × GPU Computation !!

✓ MOONCAKE WAY



Transfer is almost 50× CHEAPER!

Multi-node Systems Waste Computation Without Coordination

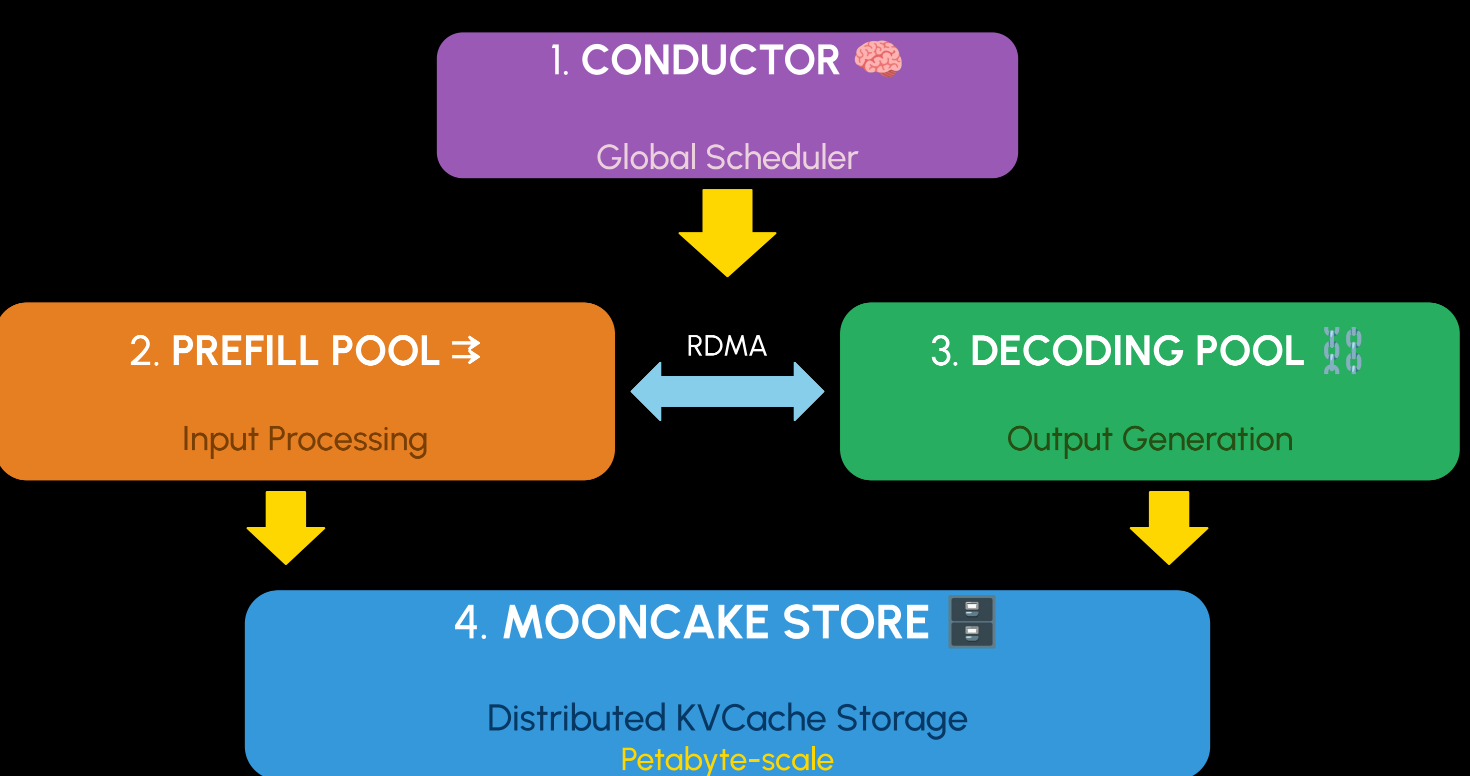


Same 50k token document computed 3 times!

Wasted Computation: 200% overhead
Cross-Node Reuse: 0%

The Mooncake Architecture

Four specialized components work together



Step 1: KVCache Reuse

Conductor searches for matching prefix cache

CONDUCTOR

Tokenizes & Searches

Search Results:

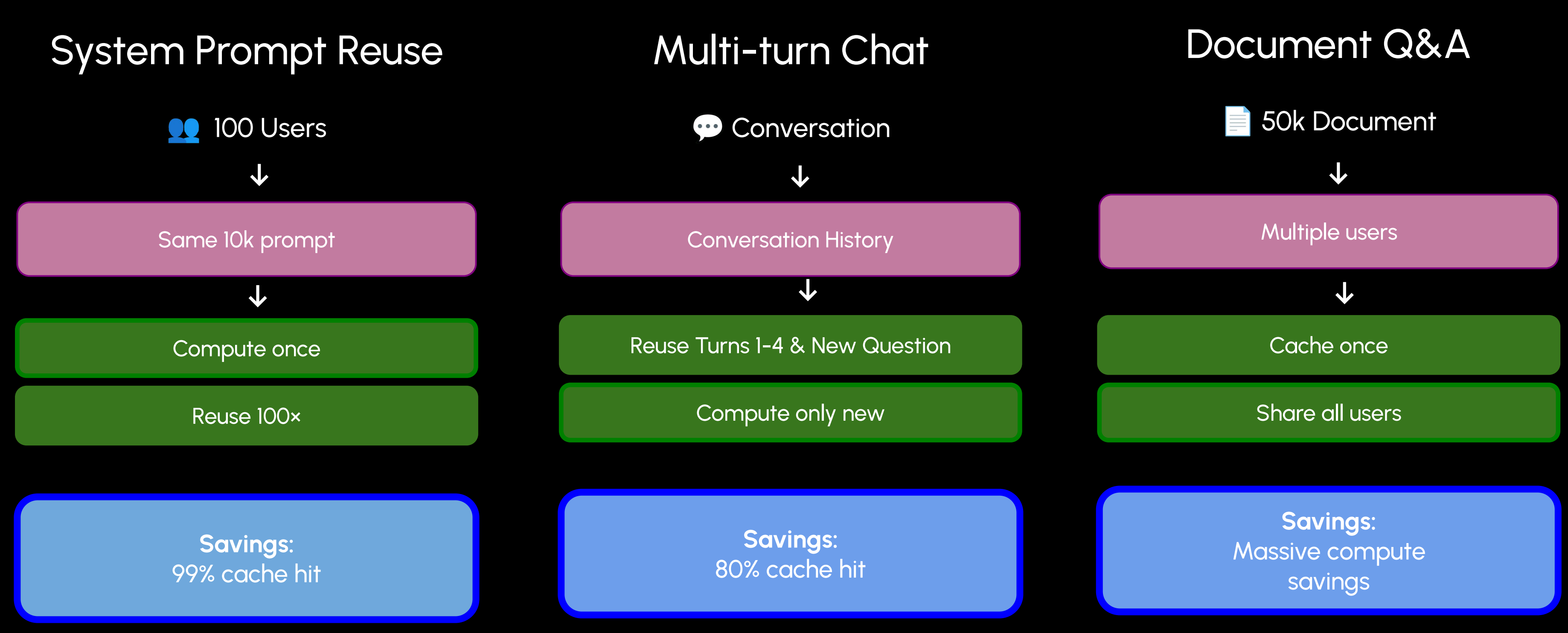
Prefix cache found!

Selects optimal prefill node

Transfer cached KVCache via RDMA

Fast network transfer

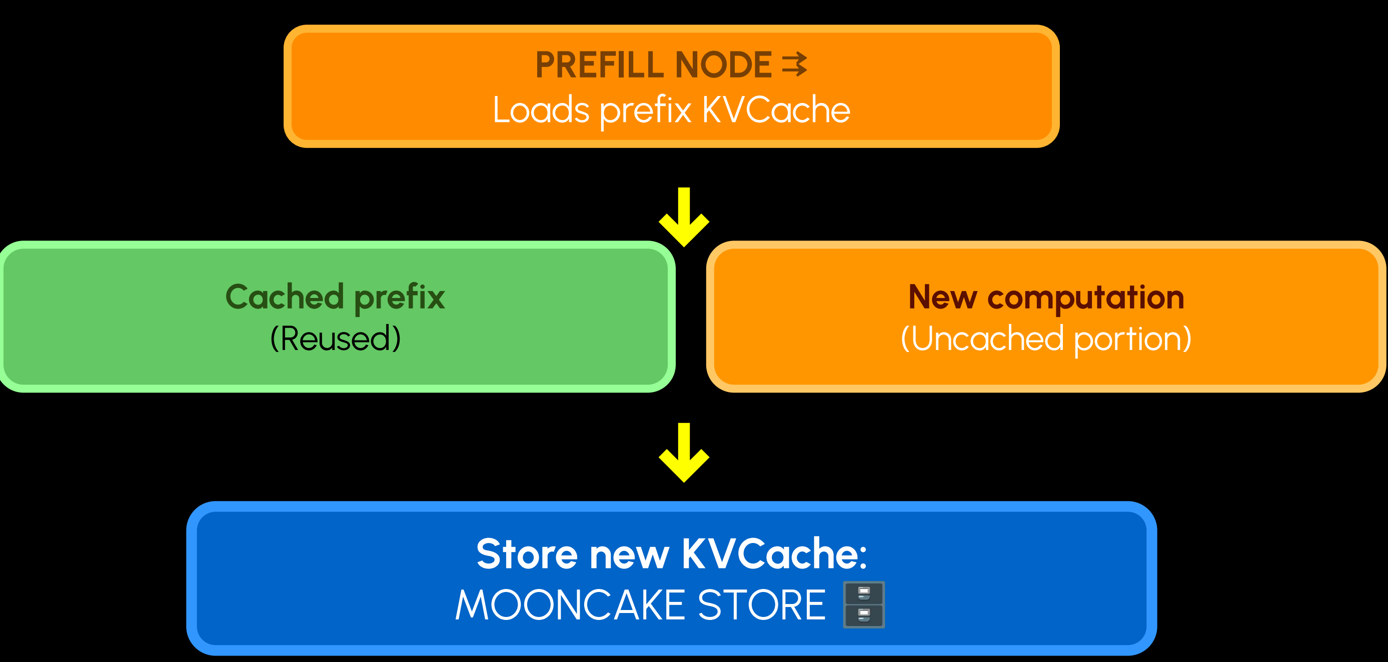
KVCache Reuse Examples



Around 50% of the KVCache tokens in the real-world workloads can be reused

Step 2: Incremental Prefill

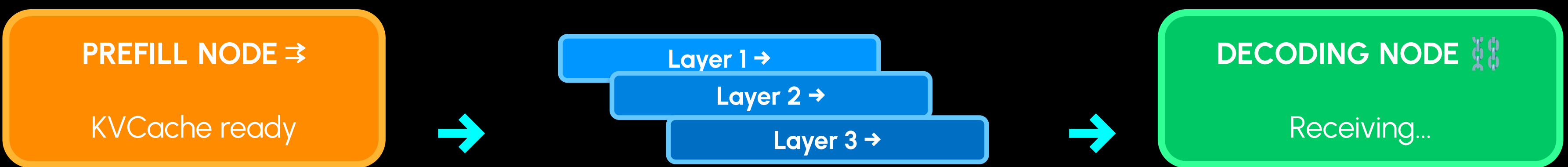
Compute ONLY uncached portion



For very long input: Uses chunked prefill (e.g., 1024 tokens/chunk)

Step 3: KVCache Transfer

Stream KVCache from Prefill → Decoding

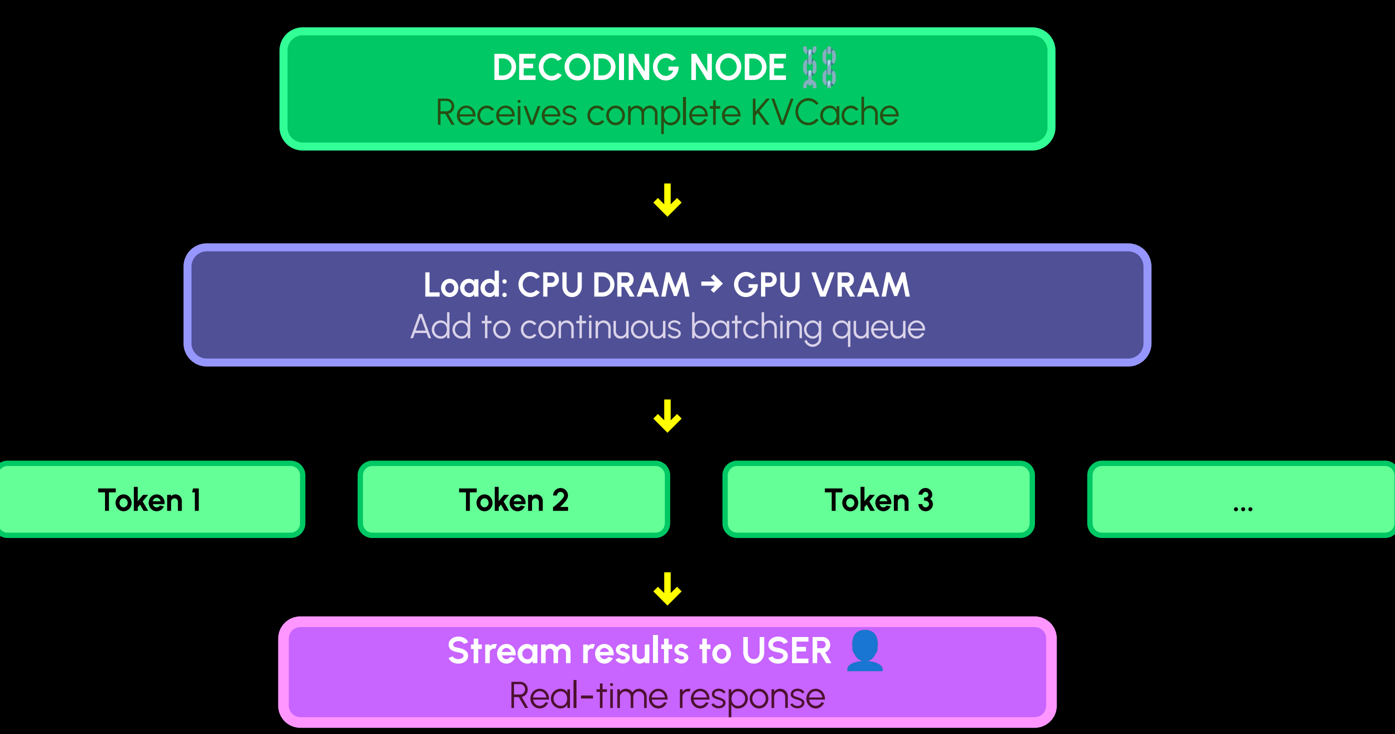


Asynchronous transfer
(overlapped with computation)

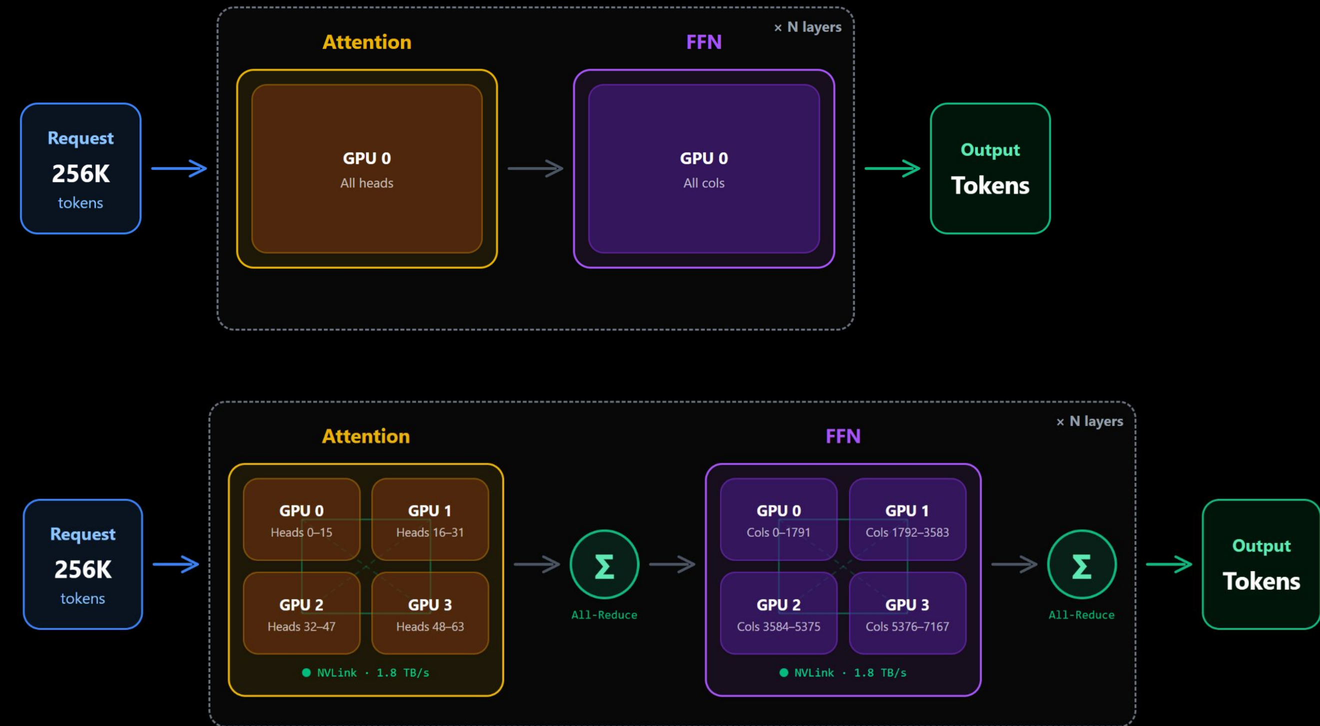
Layer-wise transfer
for better pipelining

Step 4: Decoding

Generate tokens one-by-one and stream to user

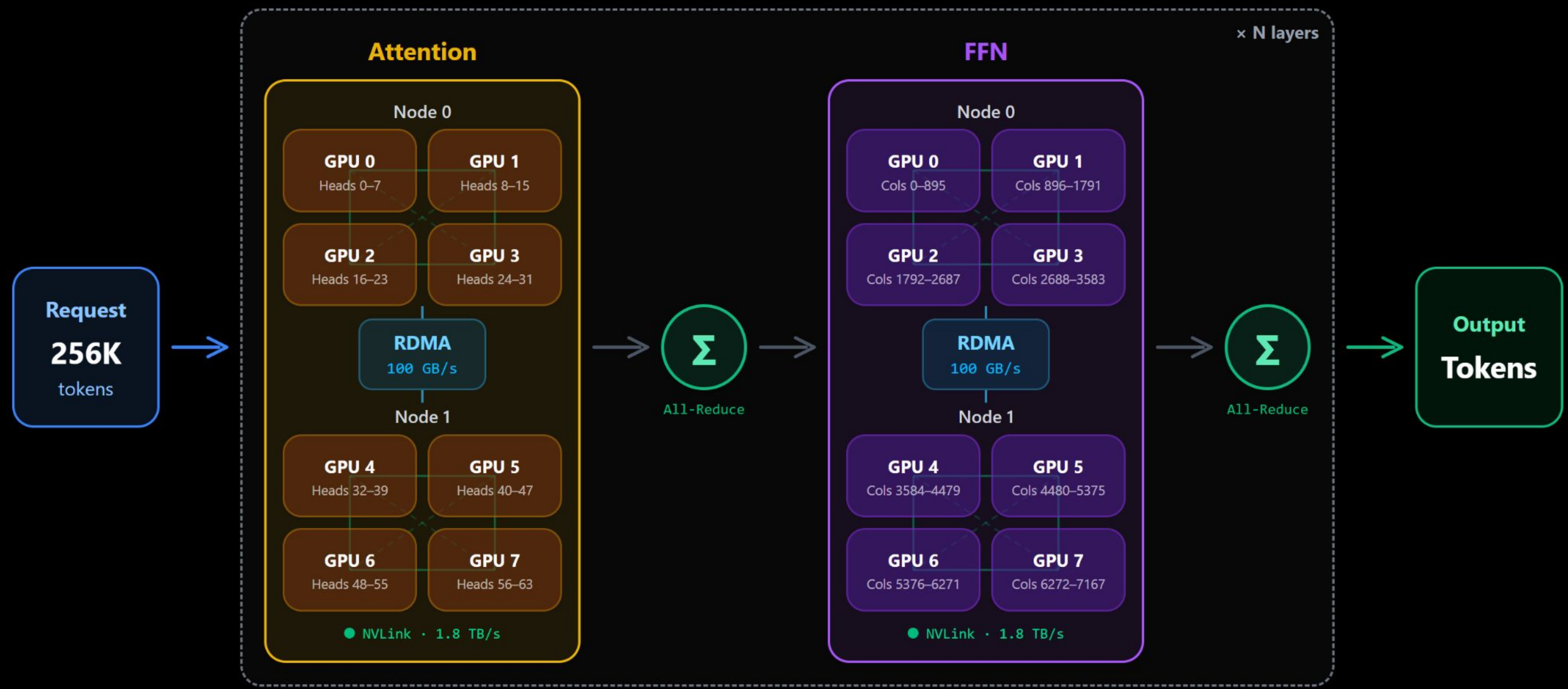


Scaling up long-context inference



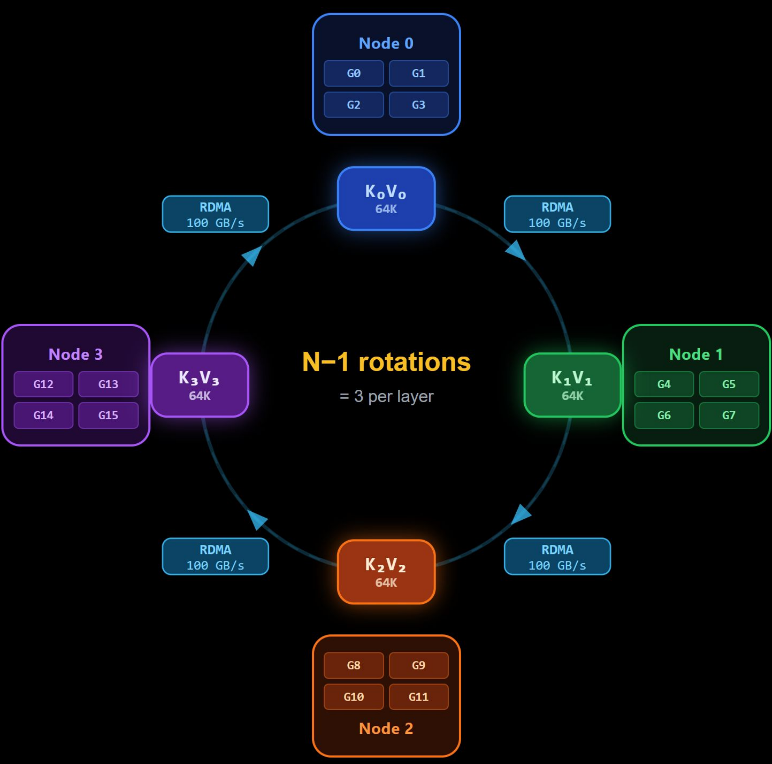
https://docs.nvidia.com/multi-node-nvlink-systems/multi-node-tuning-guide/overview.html

Scaling beyond single-node

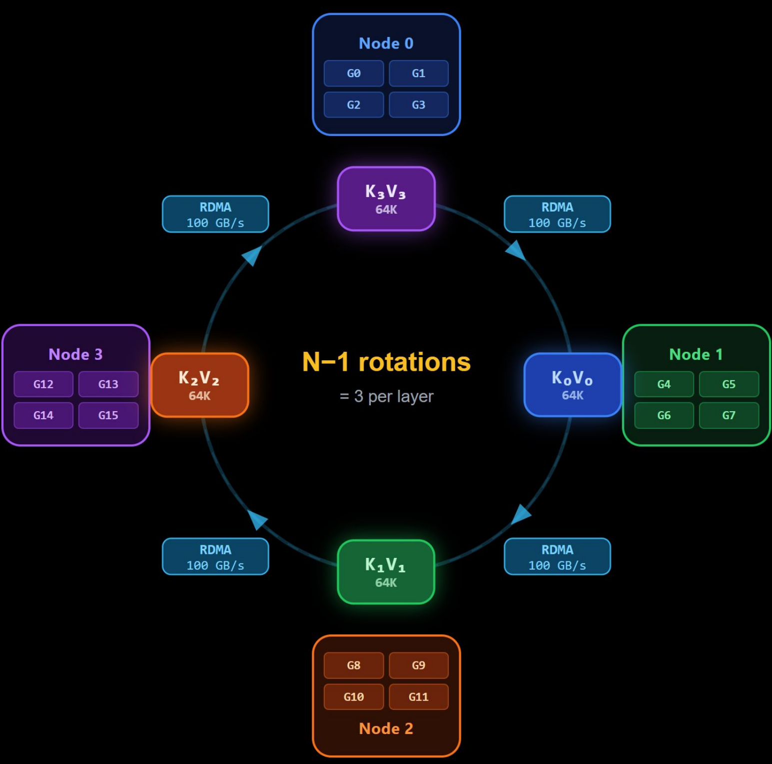


https://docs.nvidia.com/multi-node-nvlink-systems/multi-node-tuning-guide/overview.html

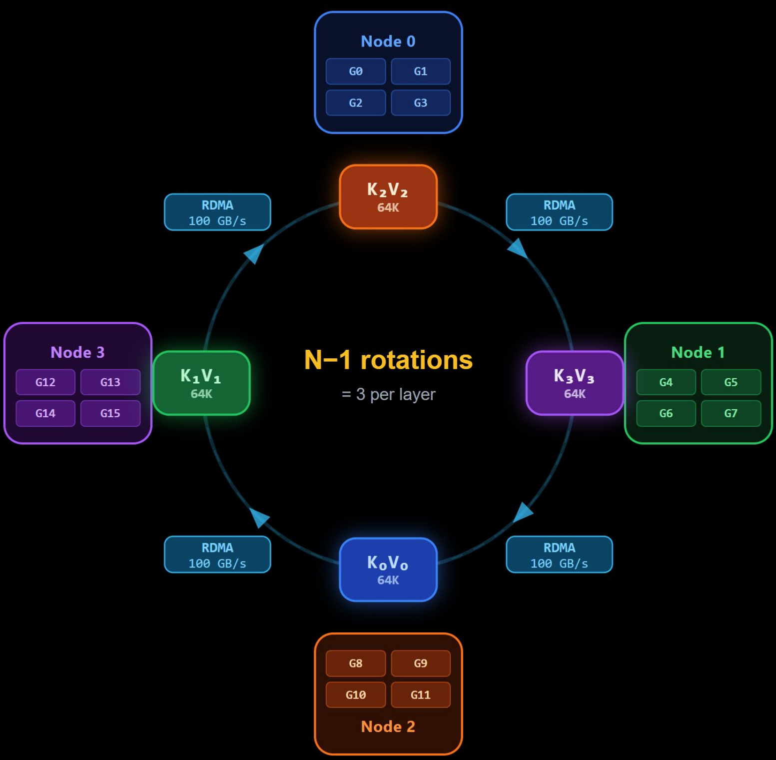
Existing solution: Sequence parallelism



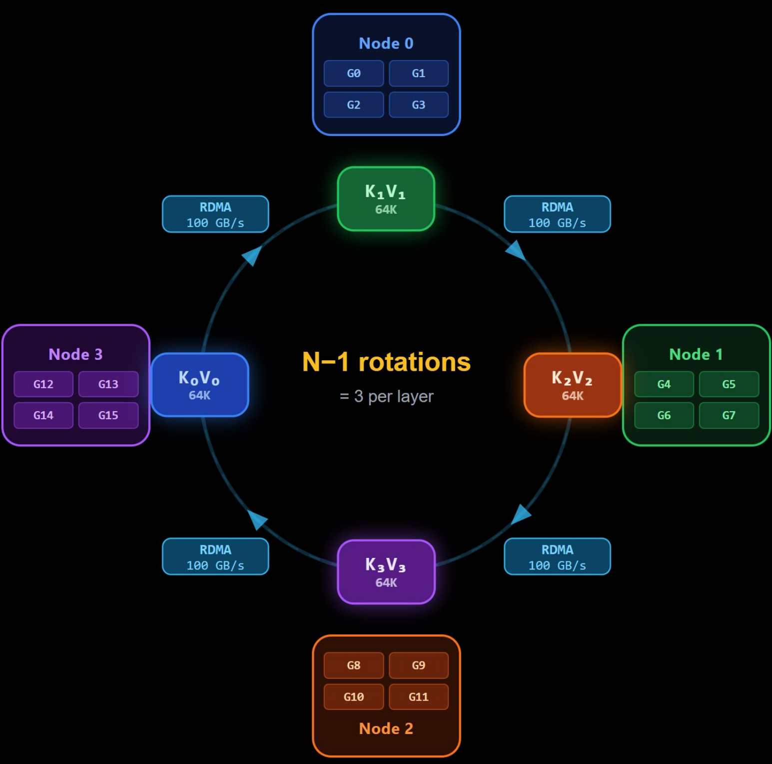
Existing solution: Sequence parallelism



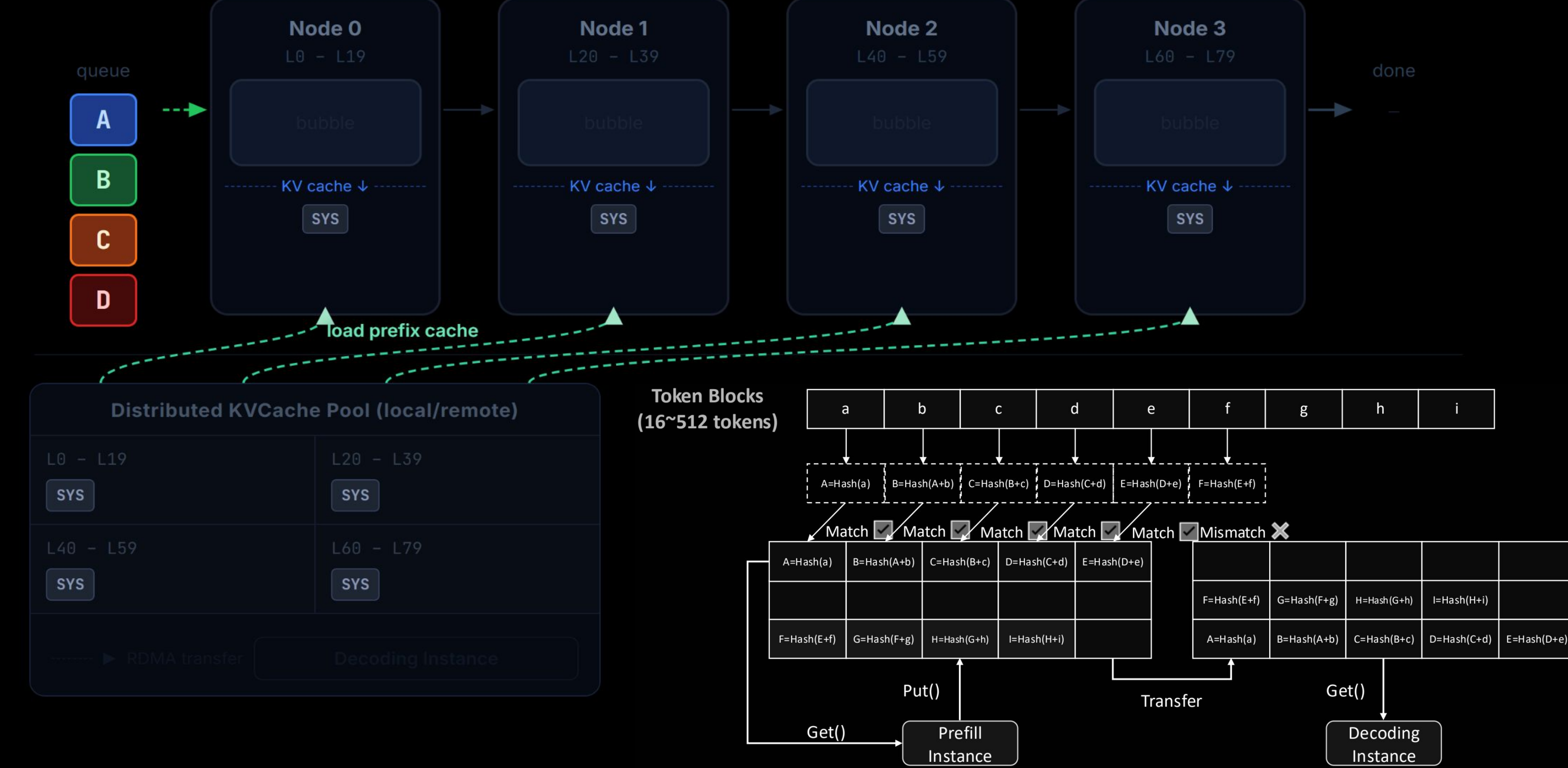
Existing solution: Sequence parallelism



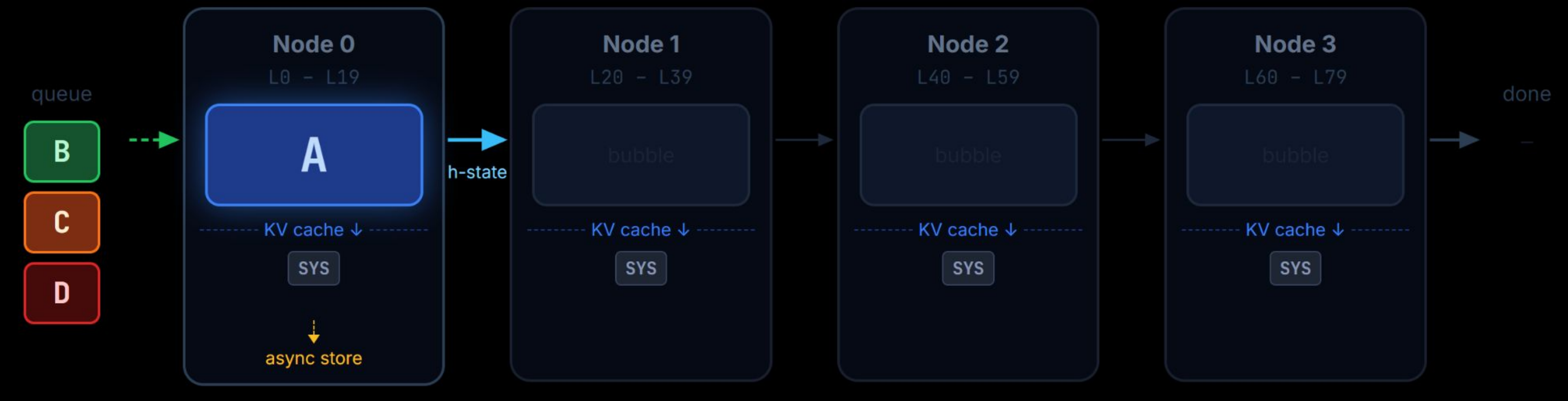
Existing solution: Sequence parallelism

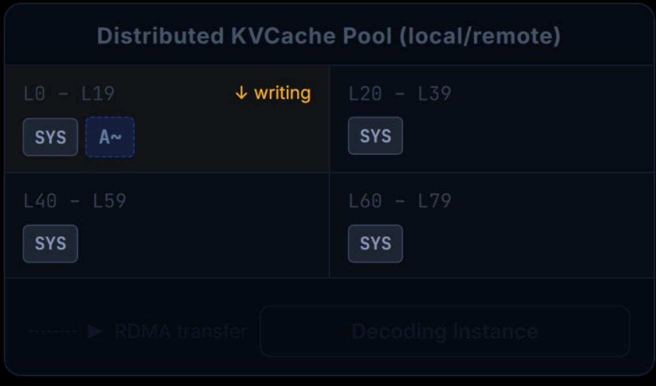


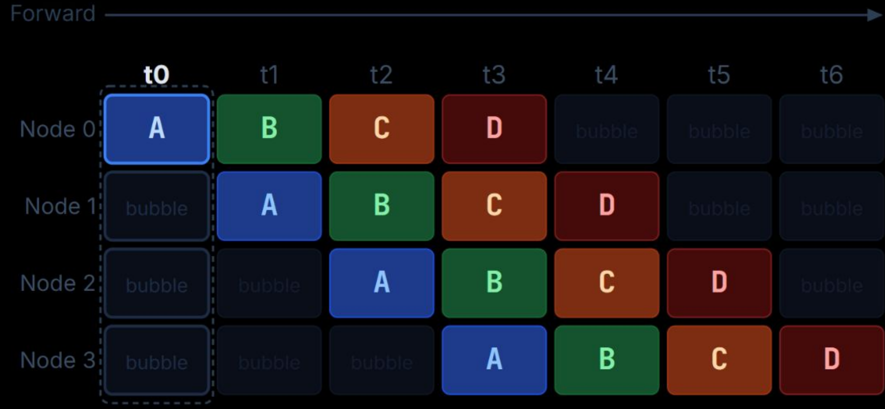
Chunked Pipeline Parallelism



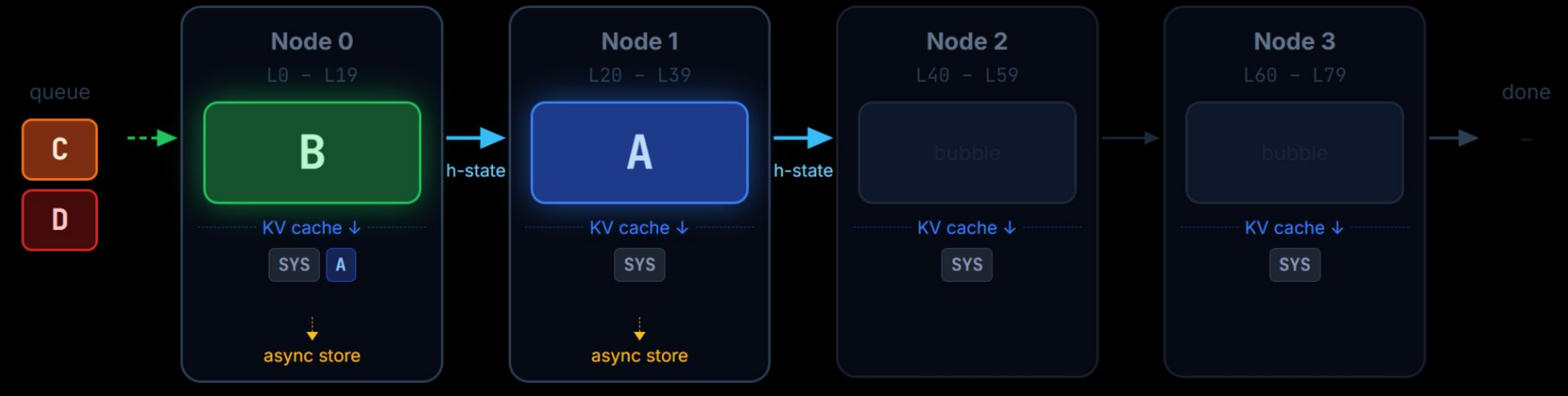
Chunked Pipeline Parallelism

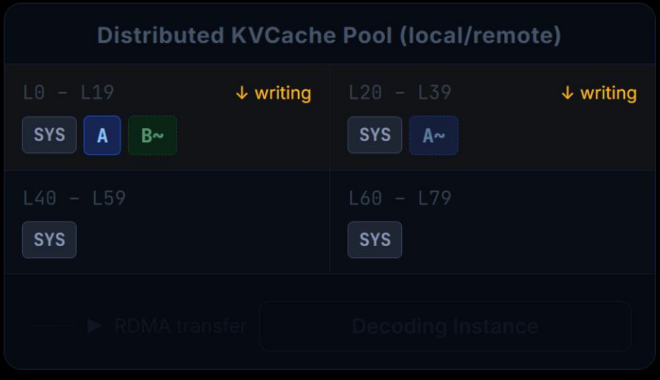


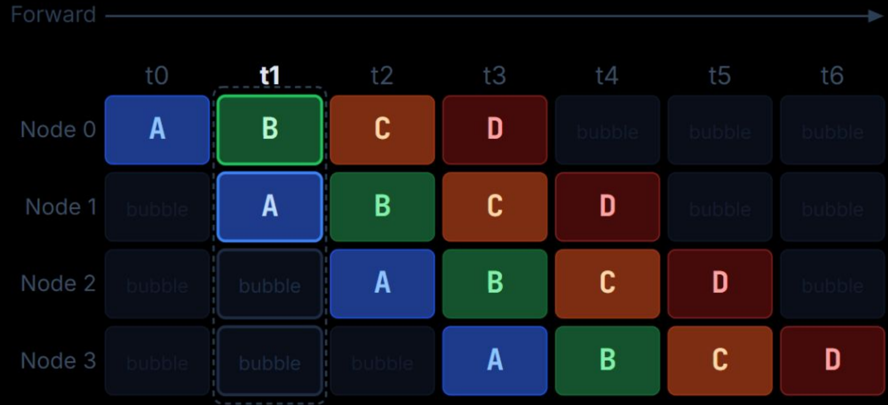




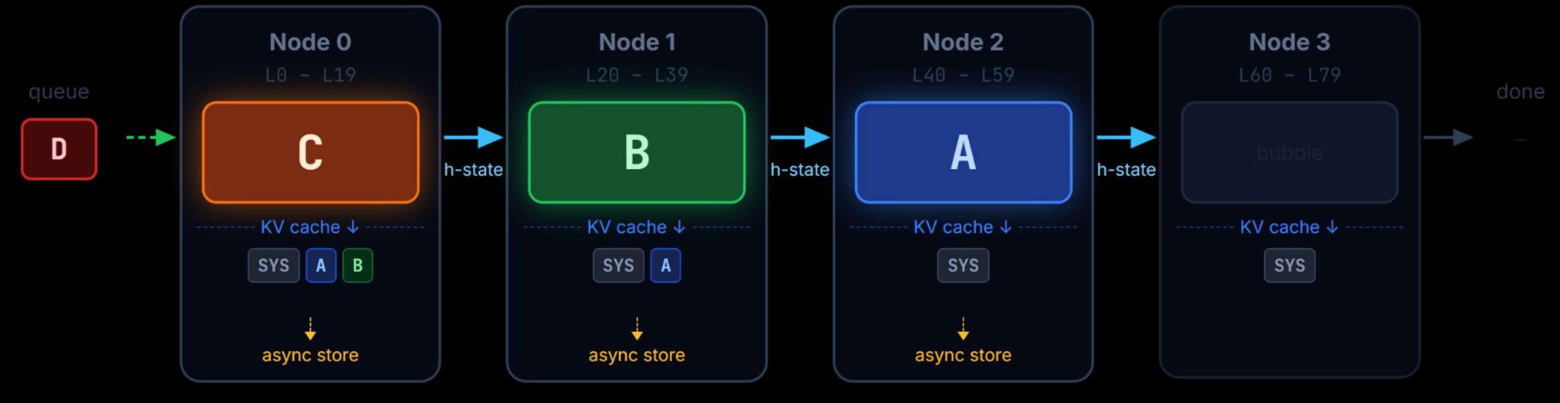
Chunked Pipeline Parallelism

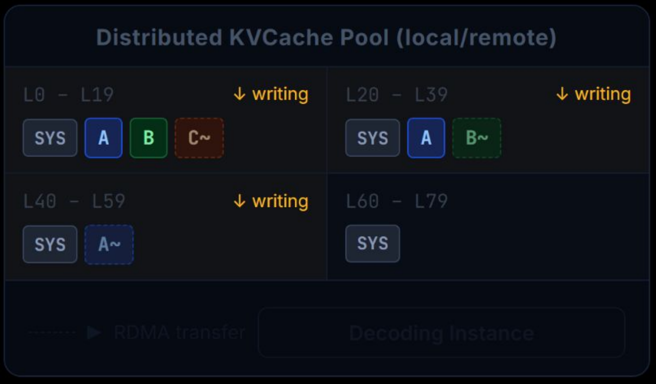


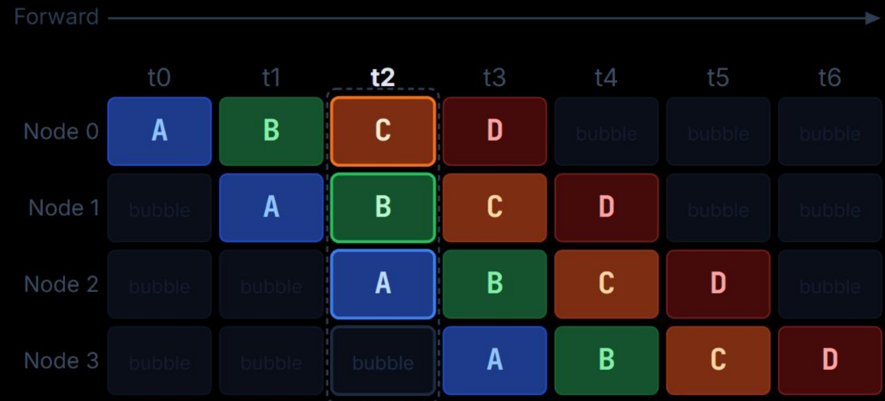




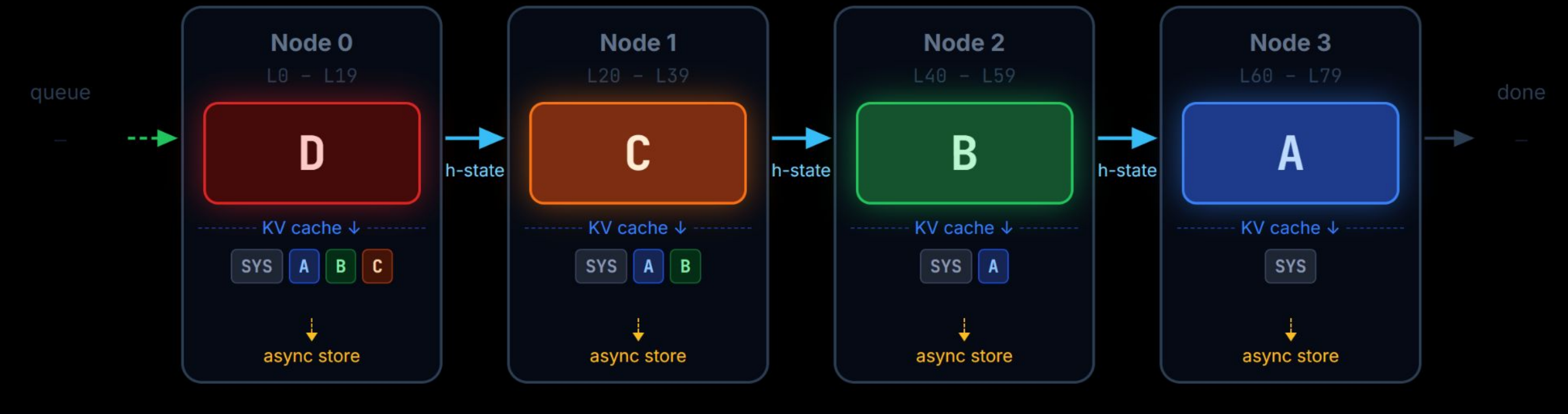
Chunked Pipeline Parallelism

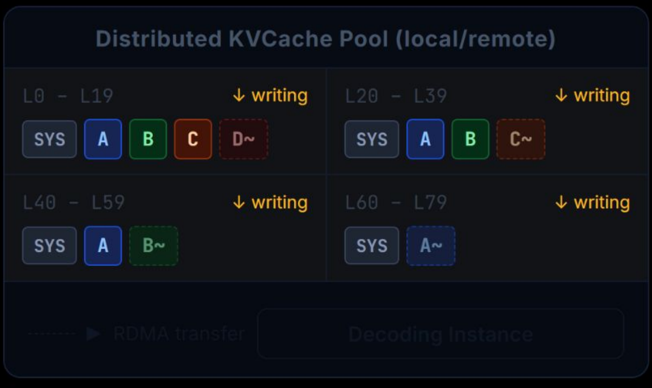




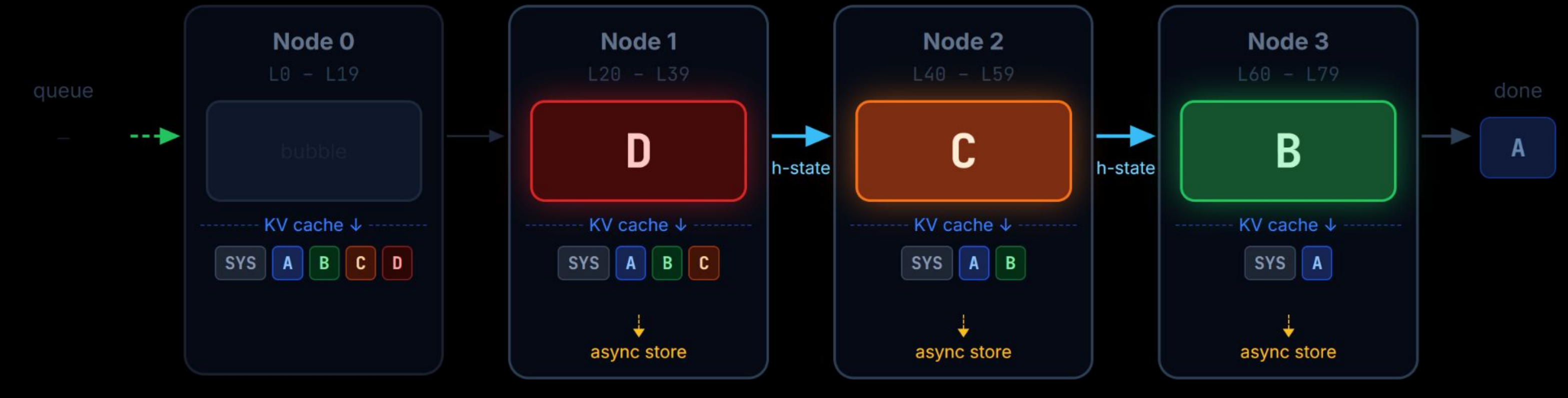


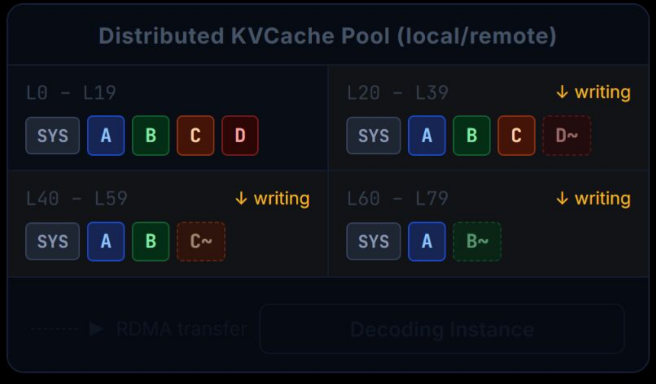
Chunked Pipeline Parallelism



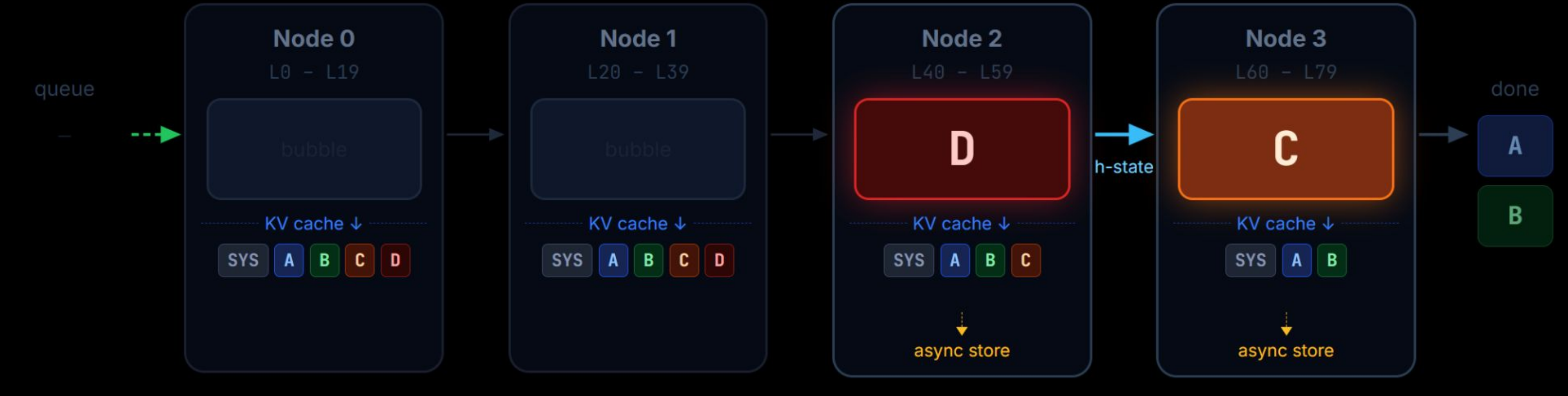


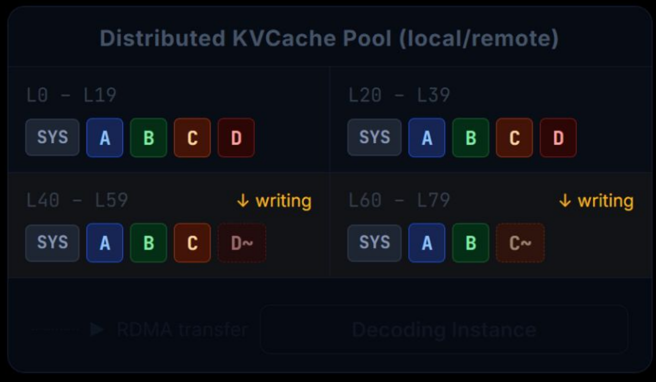
Chunked Pipeline Parallelism

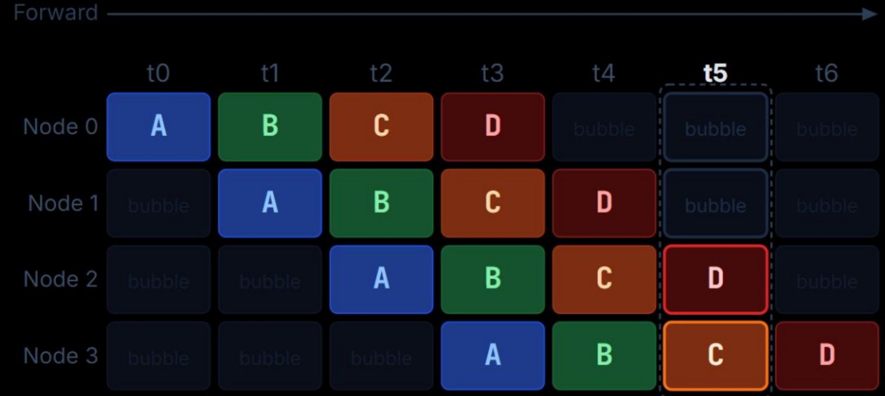




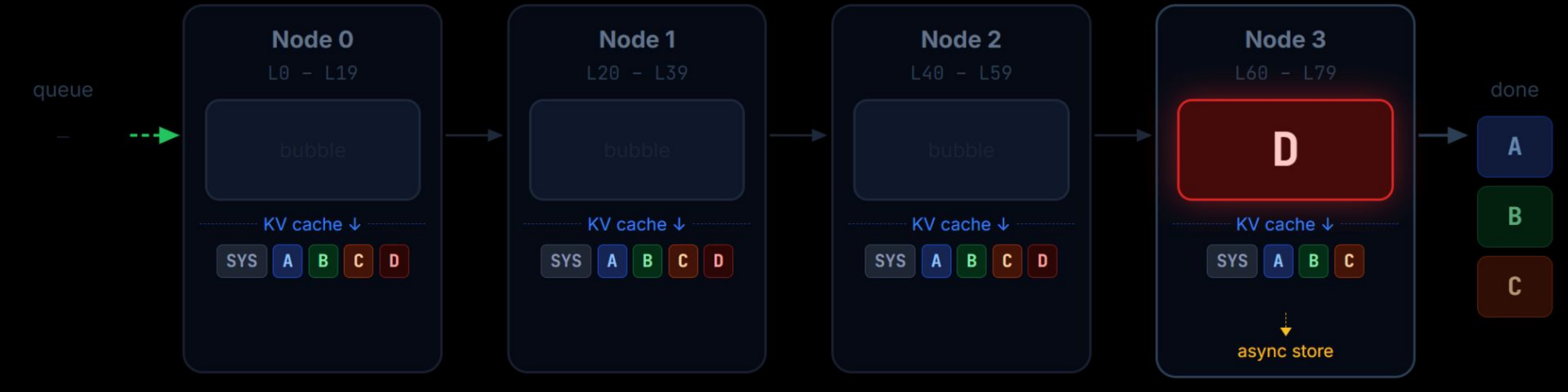
Chunked Pipeline Parallelism

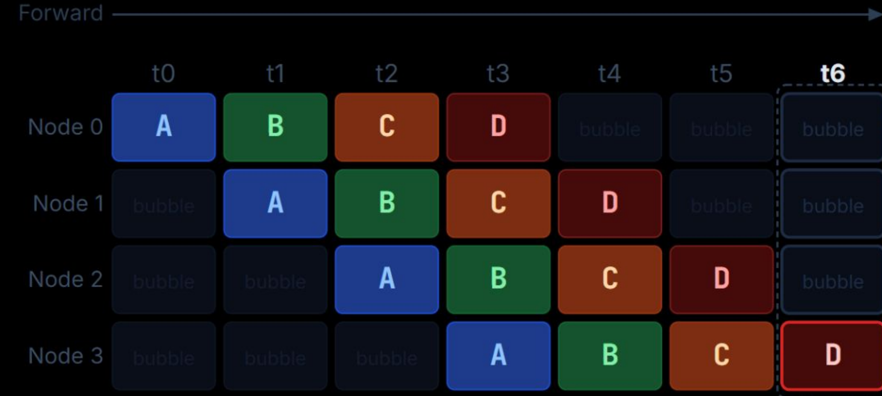




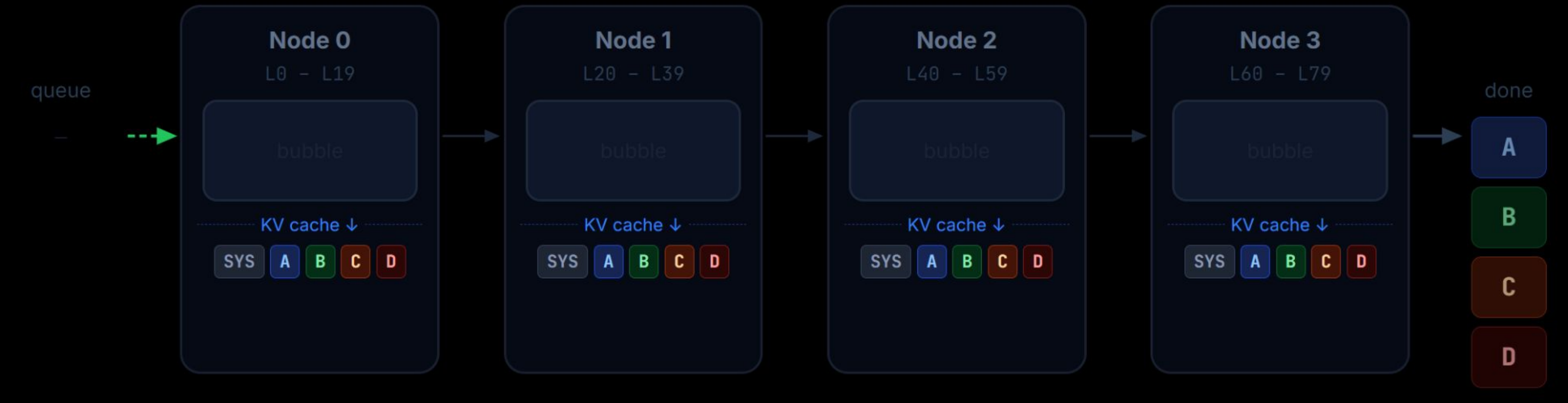


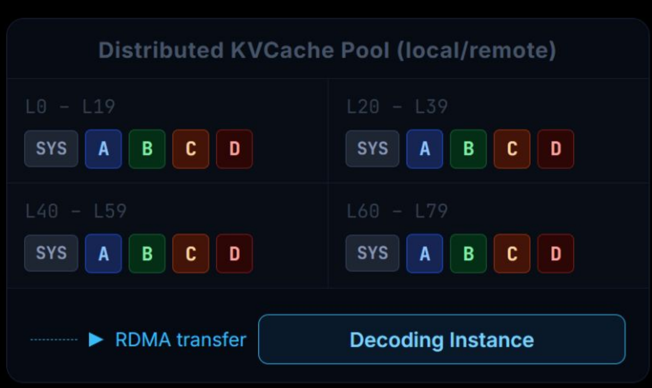
Chunked Pipeline Parallelism

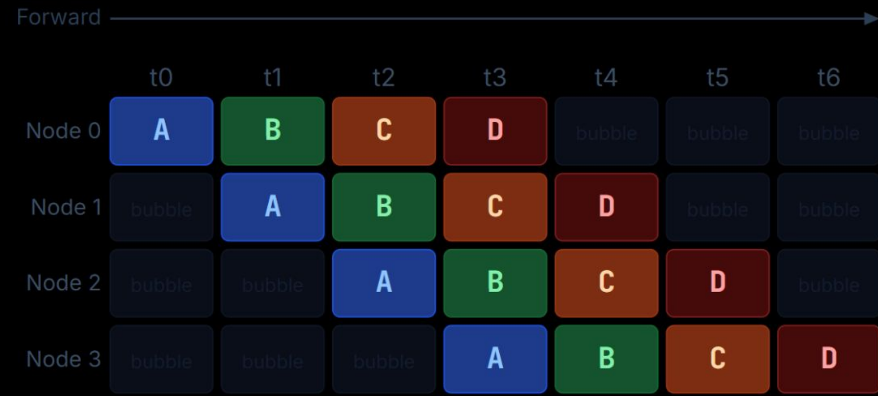




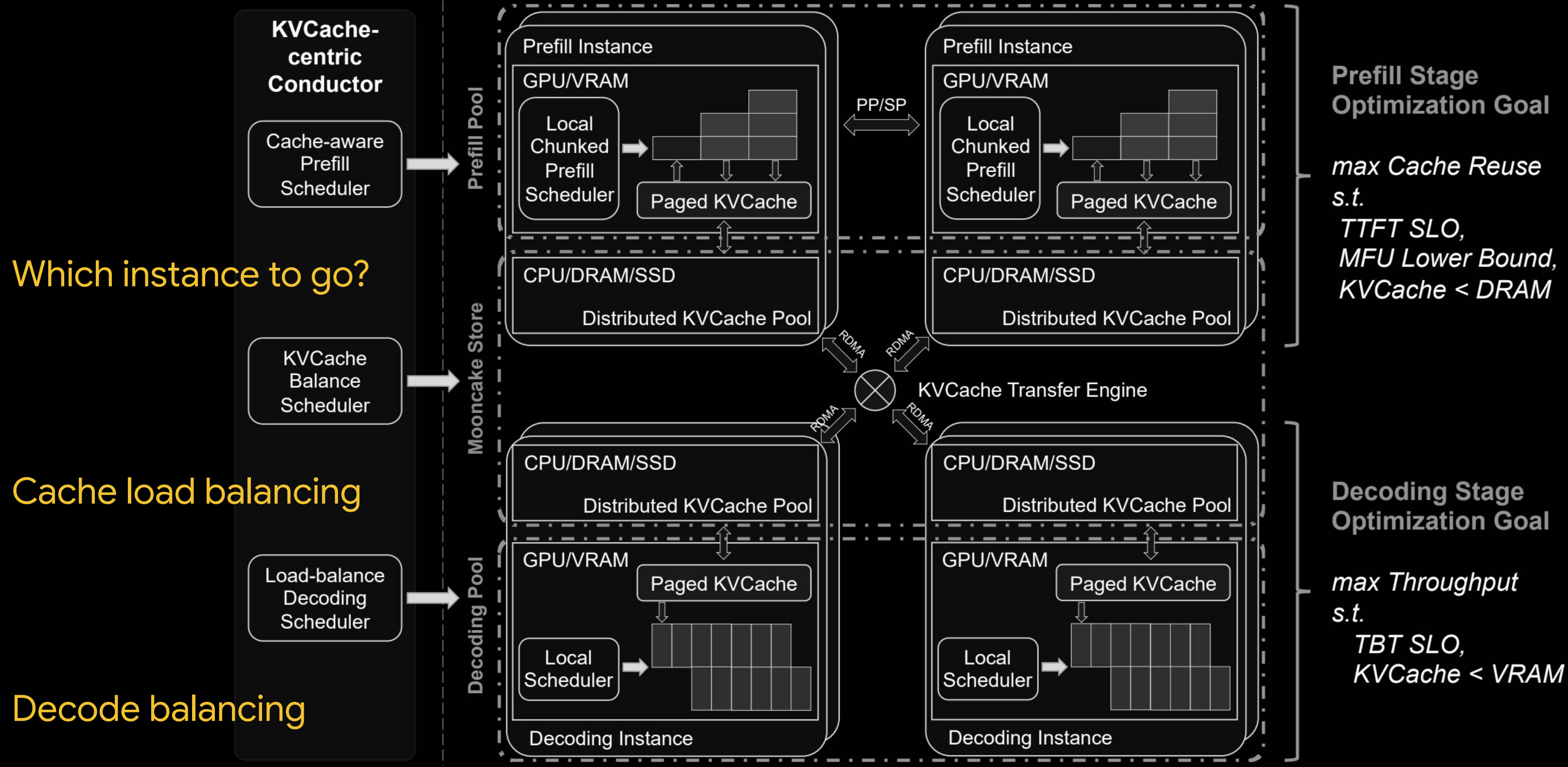
Chunked Pipeline Parallelism





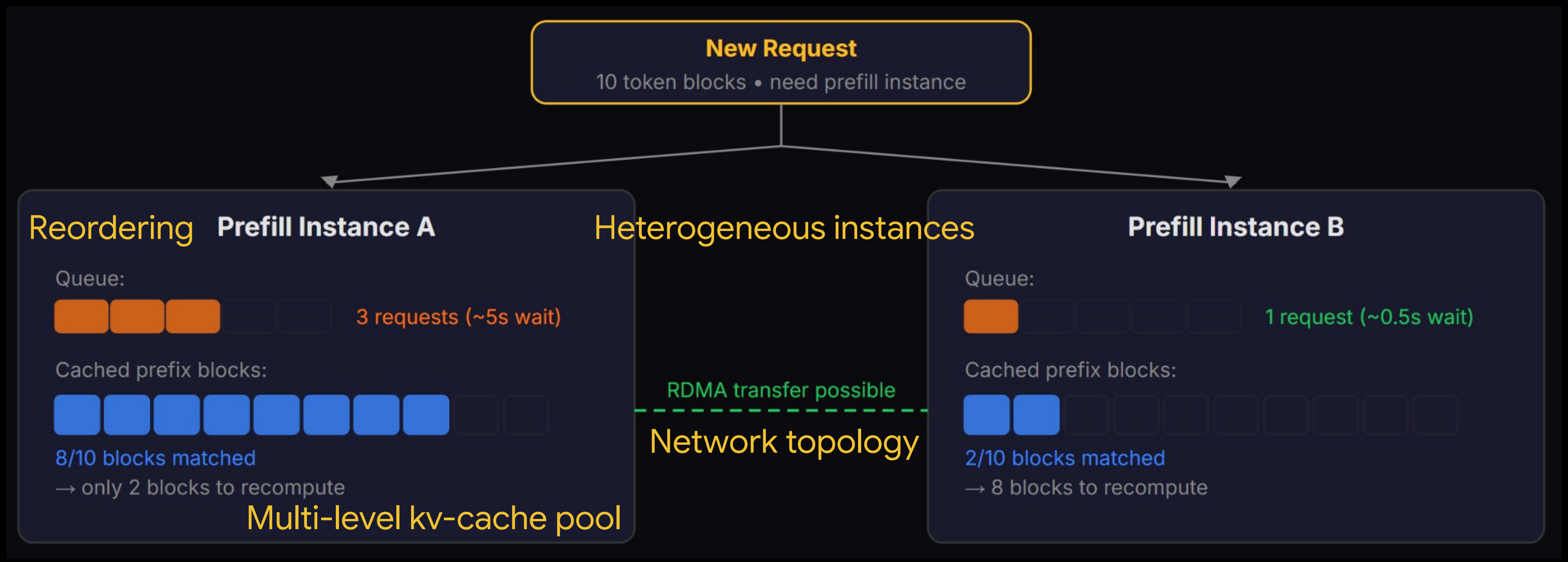


Mooncake Architecture Overview



The Scheduling Tension

Addition factor to consider?

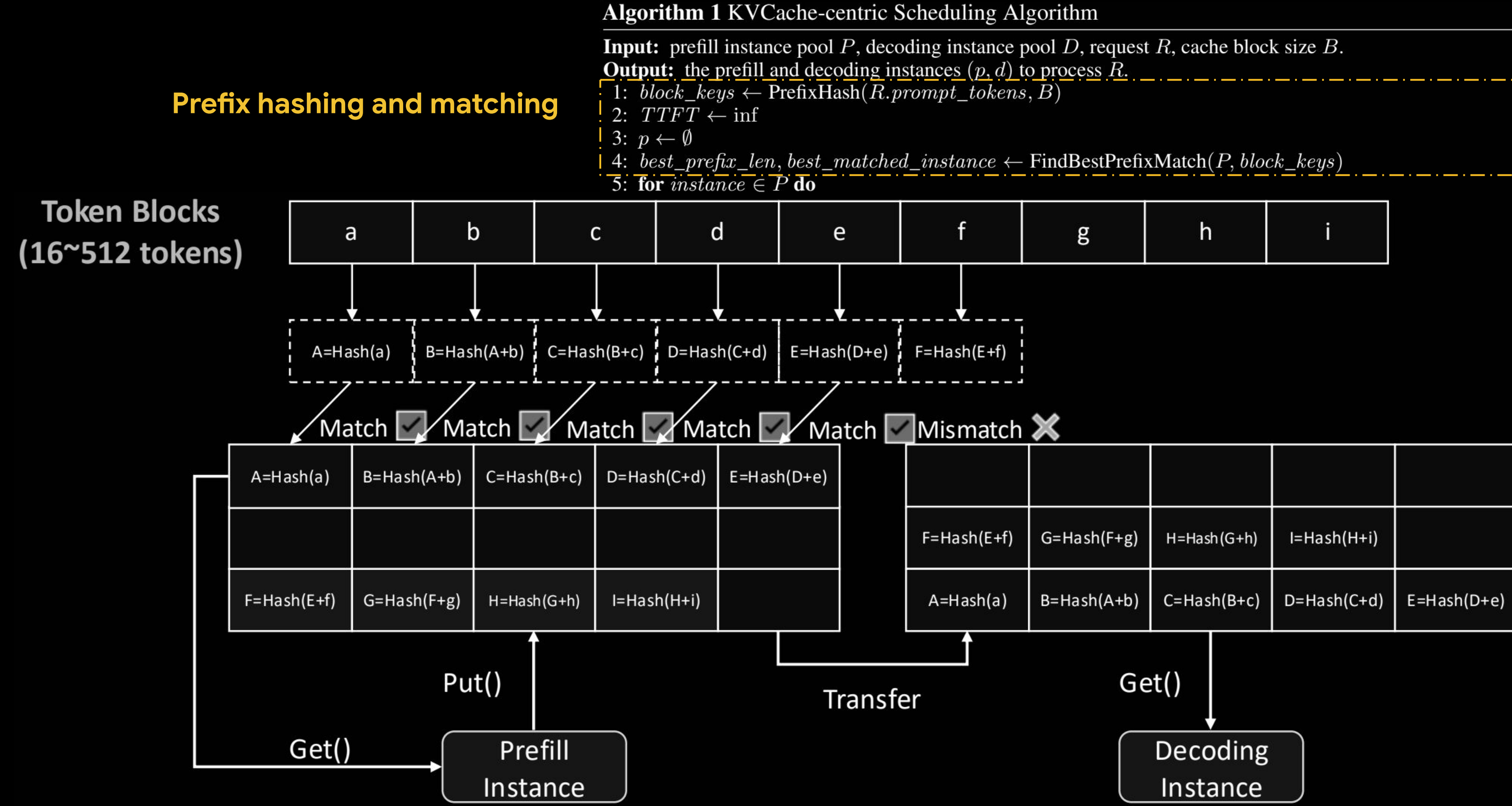


Instance A (local cache)
T_queue = 5.0s (3 requests ahead)
T_prefill = 1.0s (2 blocks)
TTFT = 6.0s

Instance B (local cache)
T_queue = 0.5s (1 requests ahead)
T_prefill = 4.0s (8 blocks)
TTFT = 4.5s

Instance B (RDMA transfer)
T_transfer = 2.0s (RDMA, 6 block)
T_queue + T_prefill = 1.5s
TTFT = 3.5s

The Scheduler Design



The Scheduler Design

Prefix hashing and matching

Compute in Parallel

Strategy A (local cache)

Manually tuned

$$\frac{best_prefix_len}{prefix_len} < \textbf{kvcache_balancing_threshold}$$

Prediction model to estimate this time

Sum of prefill exec time

SLO target

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.

1: $block_keys \leftarrow \text{PrefixHash}(R.prompt_tokens, B)$
2: $TTFT \leftarrow \text{inf}$
3: $p \leftarrow \emptyset$
4: $best_prefix_len, best_matched_instance \leftarrow \text{FindBestPrefixMatch}(P, block_keys)$
5: **for** $instance \in P$ **do**
6: $\quad prefix_len \leftarrow instance.prefix_len$
7: $\quad T_{queue} \leftarrow \text{EstimatePrefillQueueTime}(instance)$
8: $\quad$ **if** $\frac{best_prefix_len}{prefix_len} <$ **kvcache_balancing_threshold then** ▷ Cache-aware prefill scheduling
9: $\quad\quad T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt_tokens), prefix_len)$
10: $\quad\quad$ **if** $TTFT > T_{queue} + T_{prefill}$ **then**
11: $\quad\quad\quad TTFT \leftarrow T_{queue} + T_{prefill}$
12: $\quad\quad\quad p \leftarrow instance$
13: $\quad\quad$ **end if**
14: $\quad$ **else** ▷ Cache-aware and -balancing prefill scheduling
15: $\quad\quad transfer_len \leftarrow best_prefix_len - prefix_len$
16: $\quad\quad T_{transfer} \leftarrow \text{EstimateKVCacheTransferTime}(instance, best_matched_instance, transfer_len)$
17: $\quad\quad T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt_tokens), best_prefix_len)$
18: $\quad\quad$ **if** $TTFT > T_{transfer} + T_{queue} + T_{prefill}$ **then**
19: $\quad\quad\quad TTFT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$
20: $\quad\quad\quad p \leftarrow instance$
21: $\quad\quad$ **end if**
22: $\quad$ **end if**
23: **end for**
24: $d, TBT \leftarrow \text{SelectDecodingInstance}(D)$ ▷ Load-balancing decoding scheduling
25: **if** $TTFT > TTFT_SLO$ **or** $TBT > TBT_SLO$ **then**
26: $\quad$ **reject** R; **return**
27: **end if**
28: **if** $\frac{best_prefix_len}{p.prefix_len} >$ **kvcache_balancing_threshold then**
29: $\quad \text{TransferKVCache}(best_matched_instance, p)$ ▷ KVCache hot-spot migration
30: **end if**
31: **return** (p, d)

The Scheduler Design

Prefix hashing and matching

Compute in Parallel

Strategy A (local cache)

Strategy B (Remote fetch)

Challenging to predict

Dynamic network traffic, topology...

Mitigation: replicated frequently used cache

Sum of prefill exec time

SLO target

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.

1: $block_keys \leftarrow$ PrefixHash$(R.prompt_tokens, B)$
2: $TTFT \leftarrow$ inf
3: $p \leftarrow \emptyset$
4: $best_prefix_len, best_matched_instance \leftarrow$ FindBestPrefixMatch$(P, block_keys)$
5: **for** $instance \in P$ **do**
6: $prefix_len \leftarrow instance.prefix_len$
7: $T_{queue} \leftarrow$ EstimatePrefillQueueTime$(instance)$
8: **if** $\frac{best_prefix_len}{prefix_len} <$ **kvcache_balancing_threshold then** ▷ Cache-aware prefill scheduling
9: $T_{prefill} \leftarrow$ EstimatePrefillExecutionTime(len$(R.prompt_tokens)$, $prefix_len$)
10: **if** $TTFT > T_{queue} + T_{prefill}$ **then**
11: $TTFT \leftarrow T_{queue} + T_{prefill}$
12: $p \leftarrow instance$
13: **end if**
14: **else** ▷ Cache-aware and -balancing prefill scheduling
15: $transfer_len \leftarrow best_prefix_len - prefix_len$
16: $T_{transfer} \leftarrow$ EstimateKVCacheTransferTime$(instance, best_matched_instance, transfer_len)$
17: $T_{prefill} \leftarrow$ EstimatePrefillExecutionTime(len$(R.prompt_tokens)$, $best_prefix_len$)
18: **if** $TTFT > T_{transfer} + T_{queue} + T_{prefill}$ **then**
19: $TTFT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$
20: $p \leftarrow instance$
21: **end if**
22: **end if**
23: **end for**
24: $d, TBT \leftarrow$ SelectDecodingInstance(D) ▷ Load-balancing decoding scheduling
25: **if** $TTFT > TTFT_SLO$ **or** $TBT > TBT_SLO$ **then**
26: **reject** R; **return**
27: **end if**
28: **if** $\frac{best_prefix_len}{p.prefix_len} >$ **kvcache_balancing_threshold then**
29: TransferKVCache$(best_matched_instance, p)$ ▷ KVCache hot-spot migration
30: **end if**
31: **return** (p, d)

The Scheduler Design

Bottleneck? Improvement?

- Single point of failure
- Data access contention
- Threshold sensitivity
- Staled instance metadata

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.

Prefix hashing and matching

1: $block_keys \leftarrow$ PrefixHash$(R.prompt_tokens, B)$
2: $TTFT \leftarrow$ inf
3: $p \leftarrow \emptyset$
4: $best_prefix_len, best_matched_instance \leftarrow$ FindBestPrefixMatch$(P, block_keys)$
5: **for** $instance \in P$ **do**

Strategy A (local cache)

6: $prefix_len \leftarrow instance.prefix_len$
7: $T_{queue} \leftarrow$ EstimatePrefillQueueTime$(instance)$
8: **if** $\frac{best_prefix_len}{prefix_len} <$ **kvcache_balancing_threshold then** ▷ Cache-aware prefill scheduling
9: $T_{prefill} \leftarrow$ EstimatePrefillExecutionTime(len$(R.prompt_tokens)$, $prefix_len)$
10: **if** $TTFT > T_{queue} + T_{prefill}$ **then**
11: $TTFT \leftarrow T_{queue} + T_{prefill}$
12: $p \leftarrow instance$
13: **end if**
14: **else** ▷ Cache-aware and -balancing prefill scheduling

Strategy B (Remote fetch)

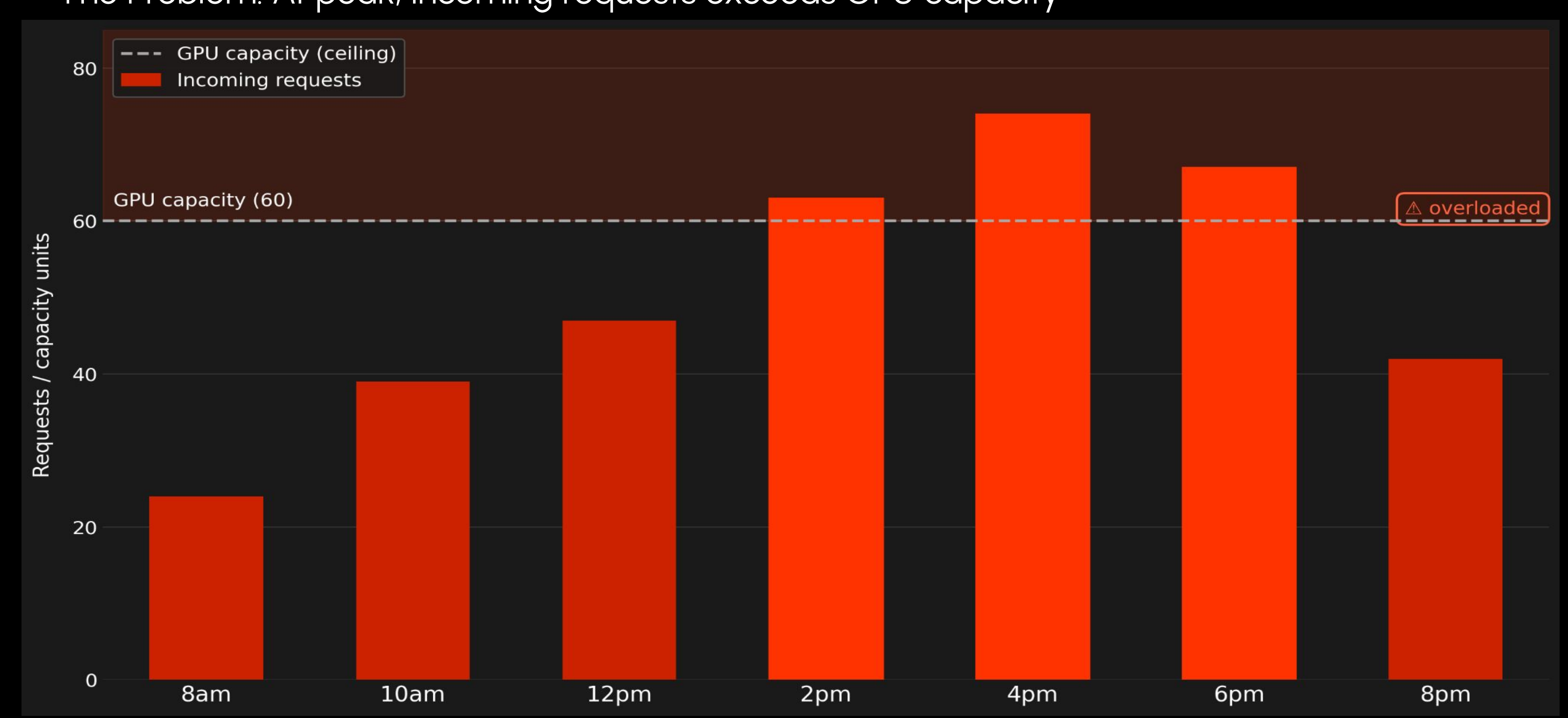
15: $transfer_len \leftarrow best_prefix_len - prefix_len$
16: $T_{transfer} \leftarrow$ EstimateKVCacheTransferTime$(instance, best_matched_instance, transfer_len)$
17: $T_{prefill} \leftarrow$ EstimatePrefillExecutionTime(len$(R.prompt_tokens)$, $best_prefix_len)$
18: **if** $TTFT > T_{transfer} + T_{queue} + T_{prefill}$ **then**
19: $TTFT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$
20: $p \leftarrow instance$
21: **end if**
22: **end if**
23: **end for**

SLO Gate and Hot-spot Migration

24: $d, TBT \leftarrow$ SelectDecodingInstance(D) ▷ Load-balancing decoding scheduling
25: **if** $TTFT > TTFT_SLO$ **or** $TBT > TBT_SLO$ **then**
26: **reject** R; **return**
27: **end if**
28: **if** $\frac{best_prefix_len}{p.prefix_len} >$ **kvcache_balancing_threshold then**
29: TransferKVCache$(best_matched_instance, p)$ ▷ KVCache hot-spot migration
30: **end if**
31: **return** (p, d)

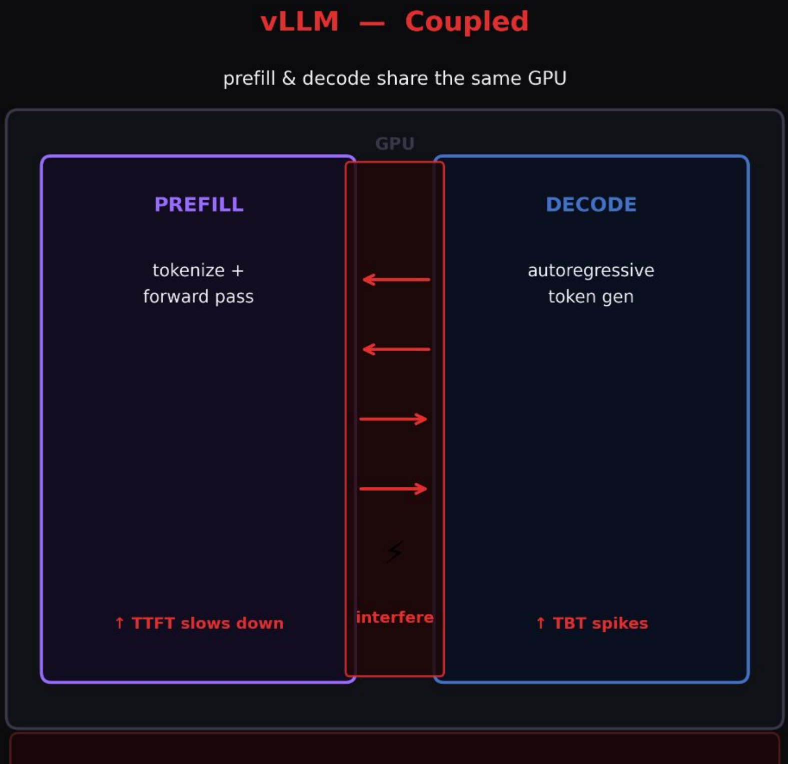
Overload-oriented Scheduling

The Problem: At peak, incoming requests exceeds GPU capacity



Overload-oriented Scheduling: Problem

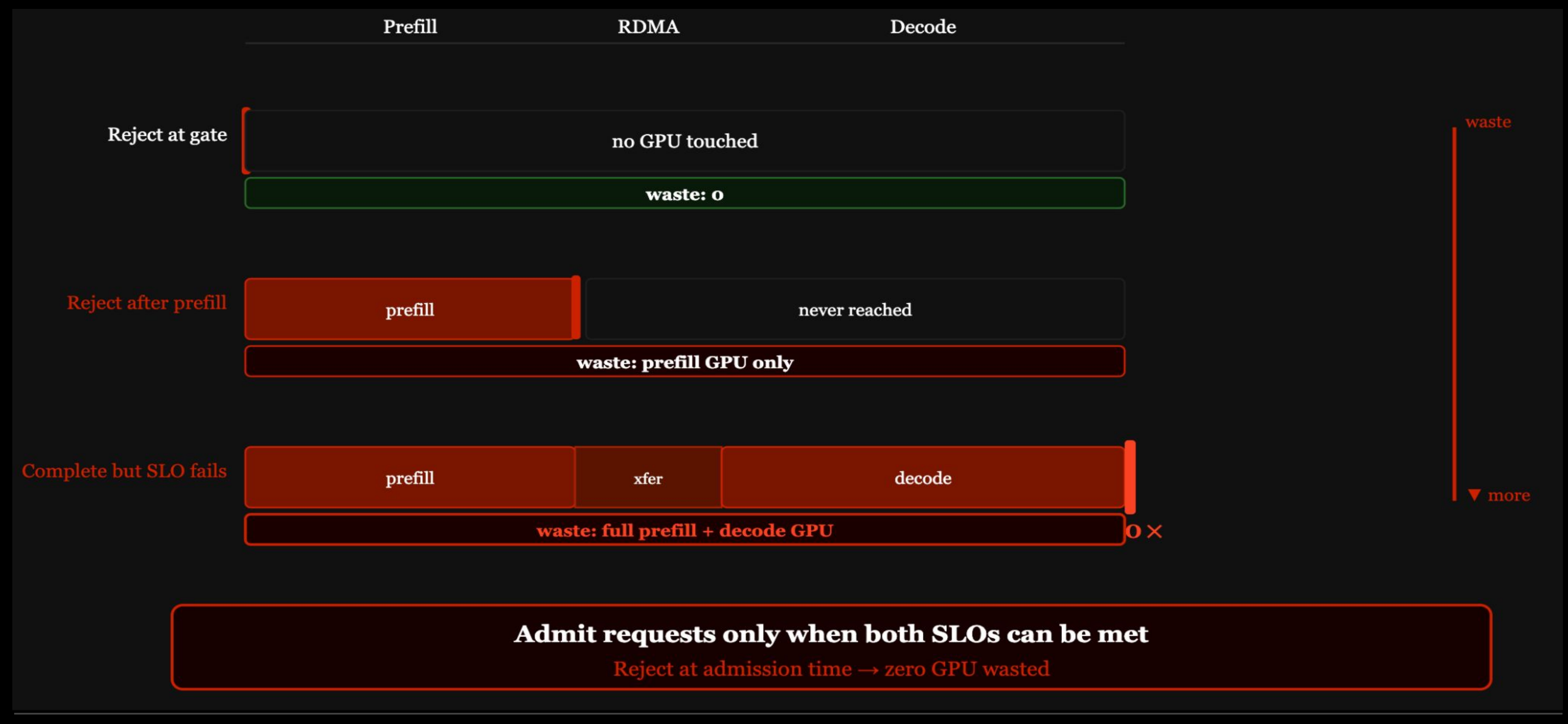
Why disaggregation matters



VS

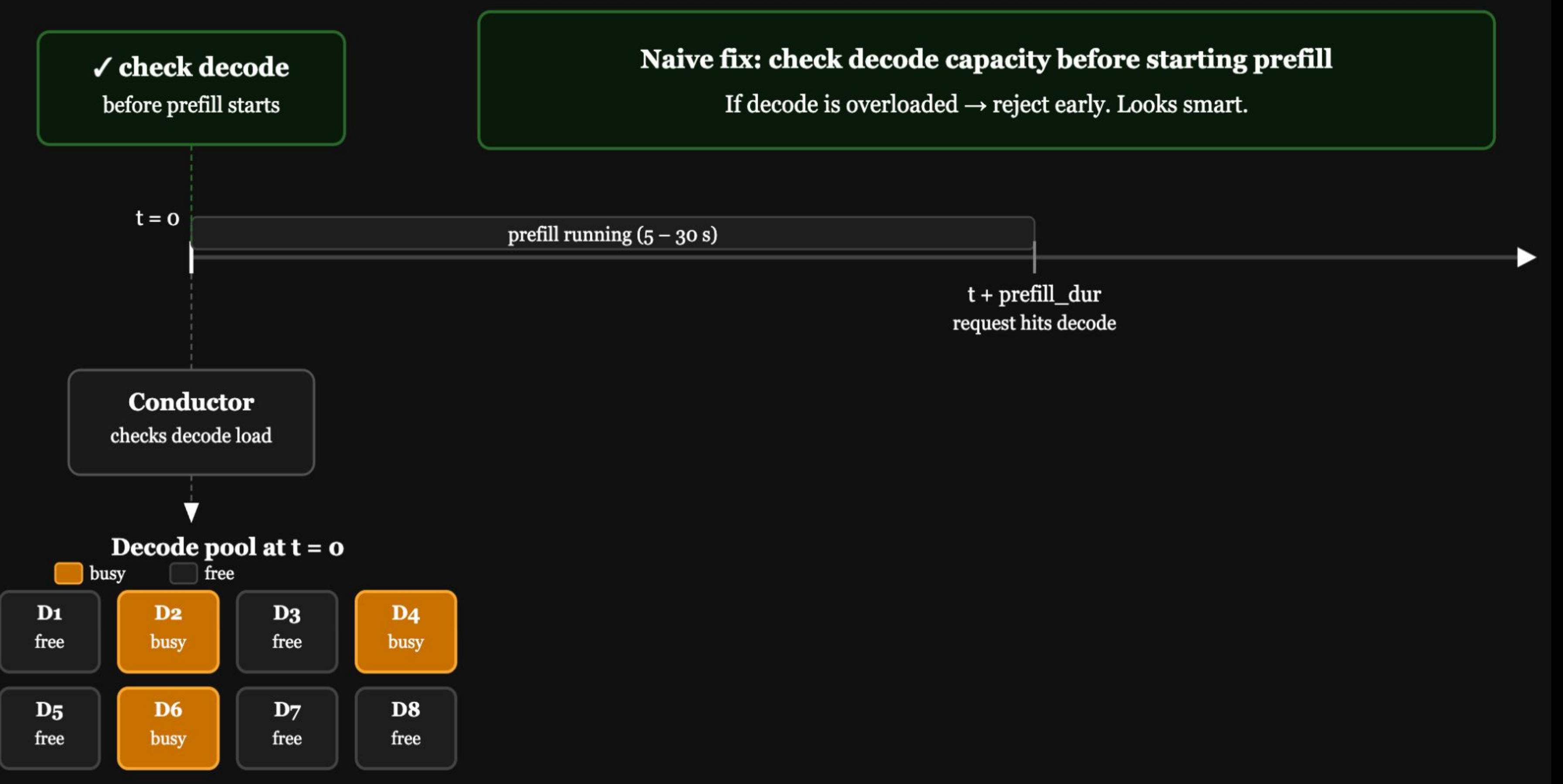
•

Overload-oriented Scheduling: Problem

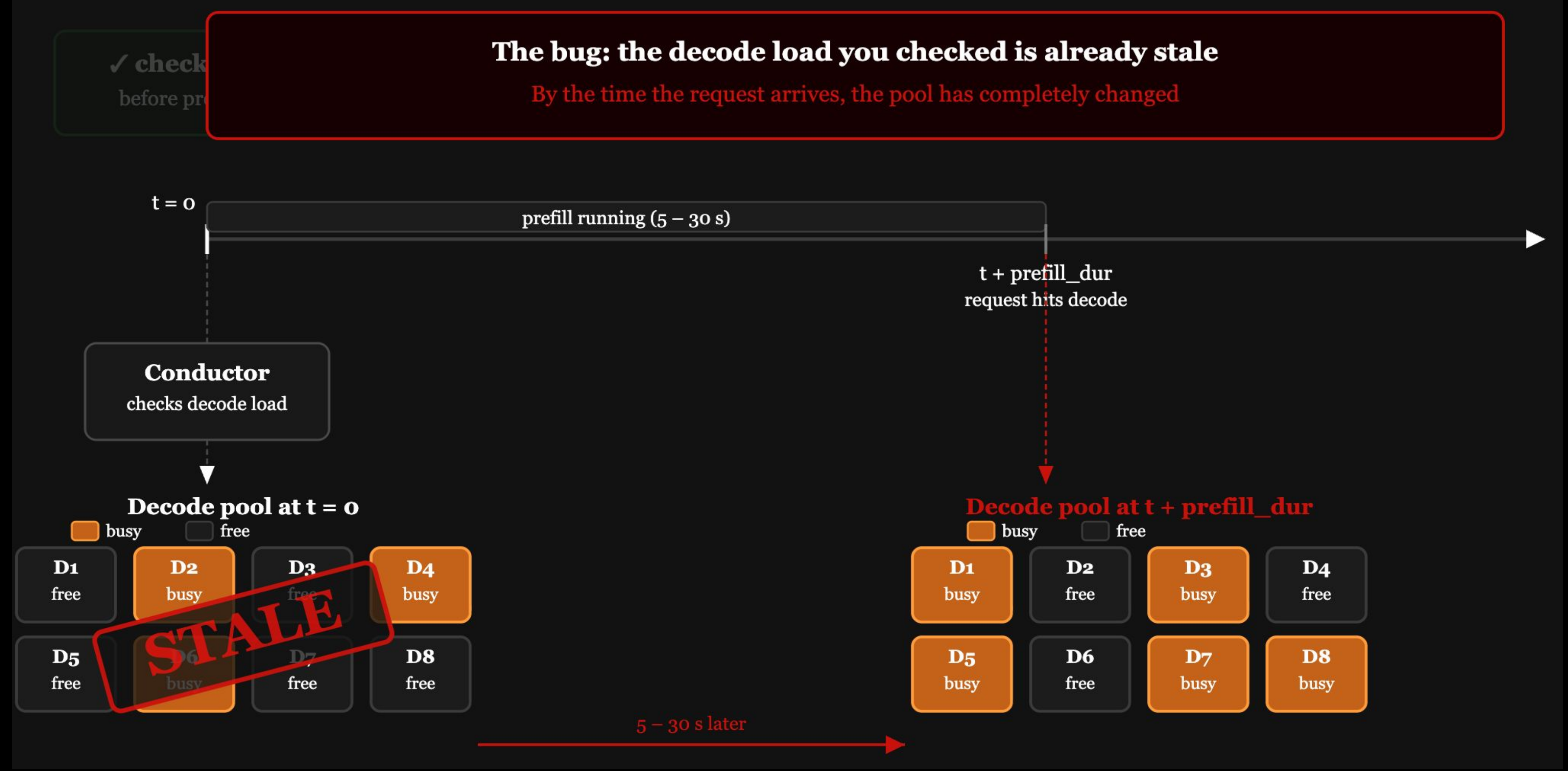


Overload-oriented Scheduling: Solution

Overload-oriented Scheduling: Solution



Overload-oriented Scheduling: Solution



Overload-oriented Scheduling: Conductor

CONDUCTOR · TWO ADMISSION GATES

Every request passes two checks before any GPU is touched

Request

Gate 1
TTFT check
× reject → HTTP 429

Gate 2
future decode check
× reject → HTTP 429

Prefill
starts here

✕
Either gate fails
HTTP 429 · 0 GPU wasted

✓
Both gates pass
Prefill starts

Overload-oriented Scheduling: Solution

Conductor: Predict Future Decode Load

New request at t = 0. Prefill = 5s. Goal: predict decode pool size at t = 5.

⚡ **New request | prefill = 5s → predict decode at t = 5 | td = 10s | capacity = 3 jobs**

IN-FLIGHT PREFILL JOBS

Job A	Job B	Job C
finishes at t = 3	finishes at t = 5	finishes at t = 8

CURRENTLY DECODING JOBS

Job X	Job Y	Job Z
started t = −8	started t = −3	started t = −1

•

Overload-oriented Scheduling: Solution

Steps 1 & 2 — Apply Two Filters

Project to t = 5. Who enters decode? Who has already left?

STEP 1 — Which prefill jobs finish by t = 5?

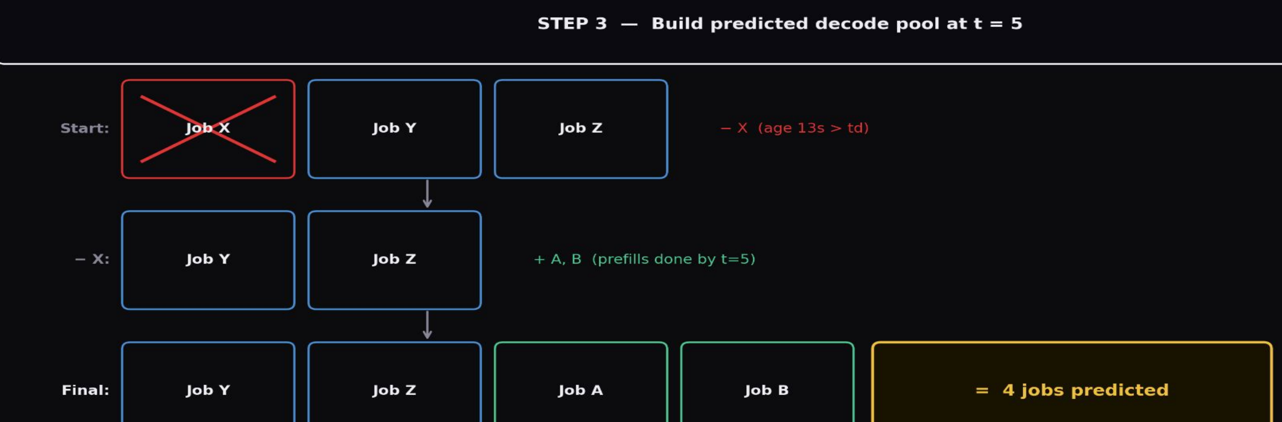
Job	Finishes	Result
Job A	finishes t = 3	✓ 3 ≤ 5 → ADD
Job B	finishes t = 5	✓ 5 = 5 → ADD
Job C	finishes t = 8	✗ 8 > 5 → SKIP

STEP 2 — Which decode jobs finish before t = 5? age = 5 − start_time. age > 10s → remove

Job	Started	Age	Result
Job X	started t = −8	age = 5−(−8) = 13s > 10s	✗ REMOVE
Job Y	started t = −3	age = 5−(−3) = 8s < 10s	✓ KEEP
Job Z	started t = −1	age = 5−(−1) = 6s < 10s	✓ KEEP

Steps 3 & 4 — Compute Pool → Reject

Build the predicted pool, then check against capacity.



STEP 4 — predicted load (4) > capacity (3)

× REJECT — HTTP 429

0 GPU wasted — rejected before prefill starts

coupled system would waste 5s of prefill compute before discovering the overload

Evaluation

Two main questions:

- Does Mooncake outperform existing LLM inference systems in real-world scenarios?

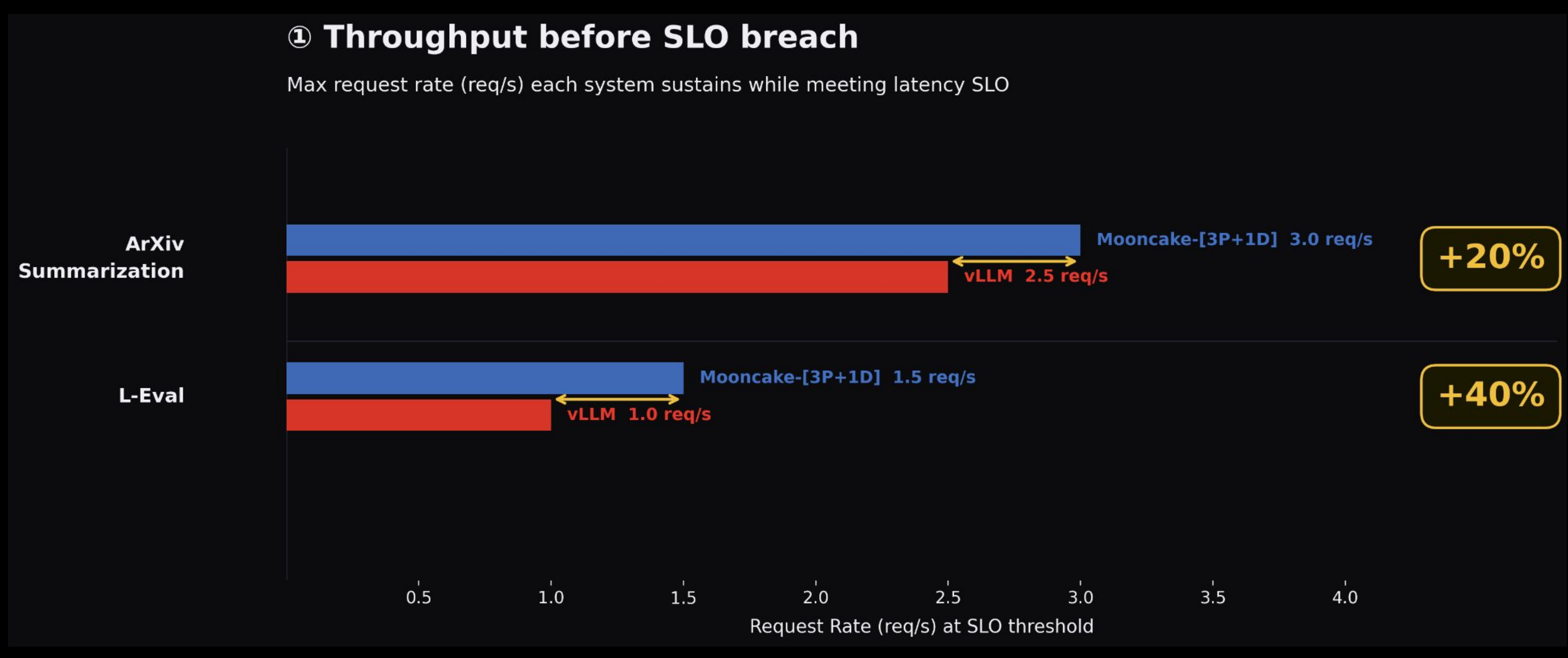
- Compared to conventional prefix caching methods, does the design of Mooncake Store significantly improve Mooncake performance?

Central idea: ***KV Cache-centric disaggregation beats vLLM in goodput.***

Evaluation: Set Up Overview

Category	Detail
Model	Dummy LLaMA3-70B
Testbed	16 nodes · 8× A800 GPUs · 4× 200 Gbps RDMA NICs
Baseline	vLLM vLLM + prefix caching + chunked prefill
Workload	3 traces (real + synthetic) resembling online request distribution
Metric	Effective request capacity · GPU cost

Experiment 1: Results

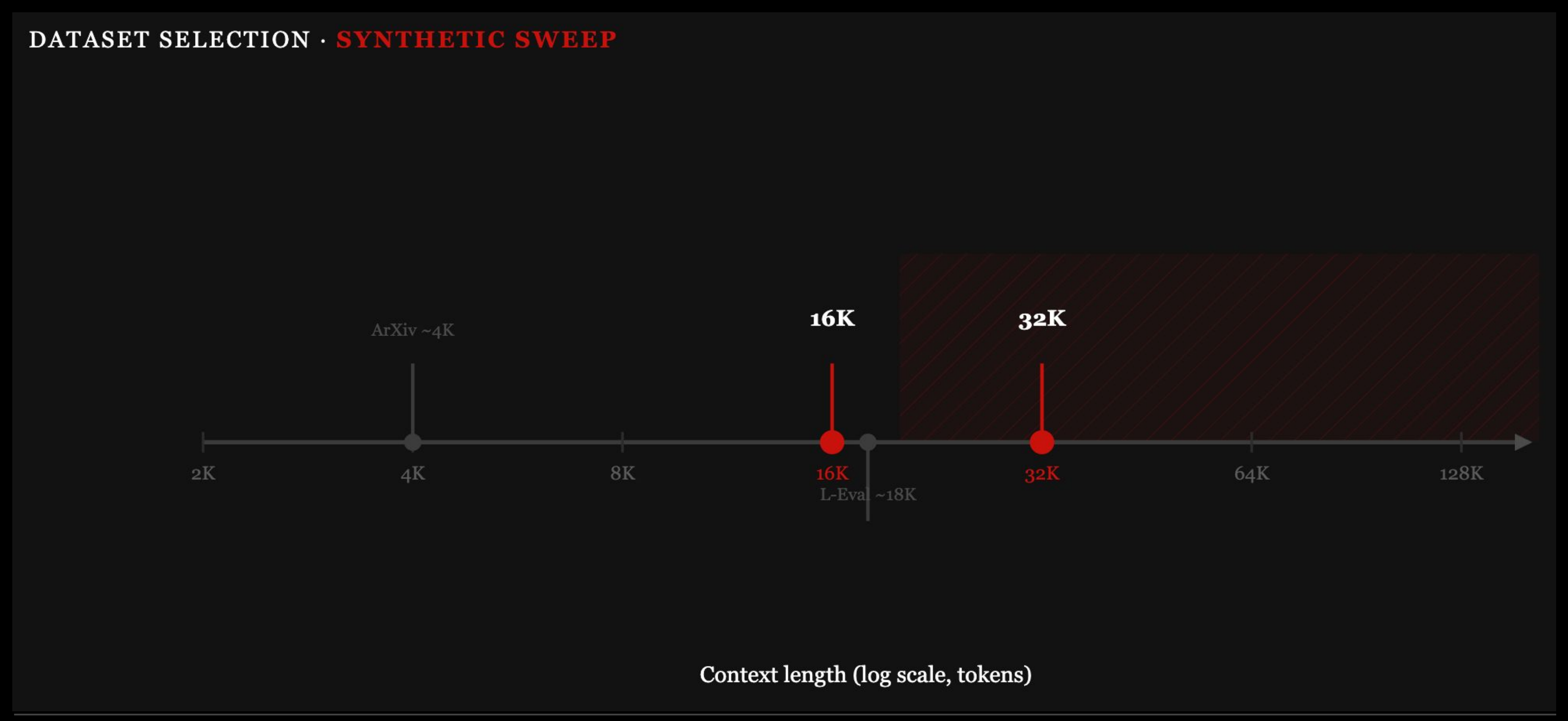


Experiment 1: Results

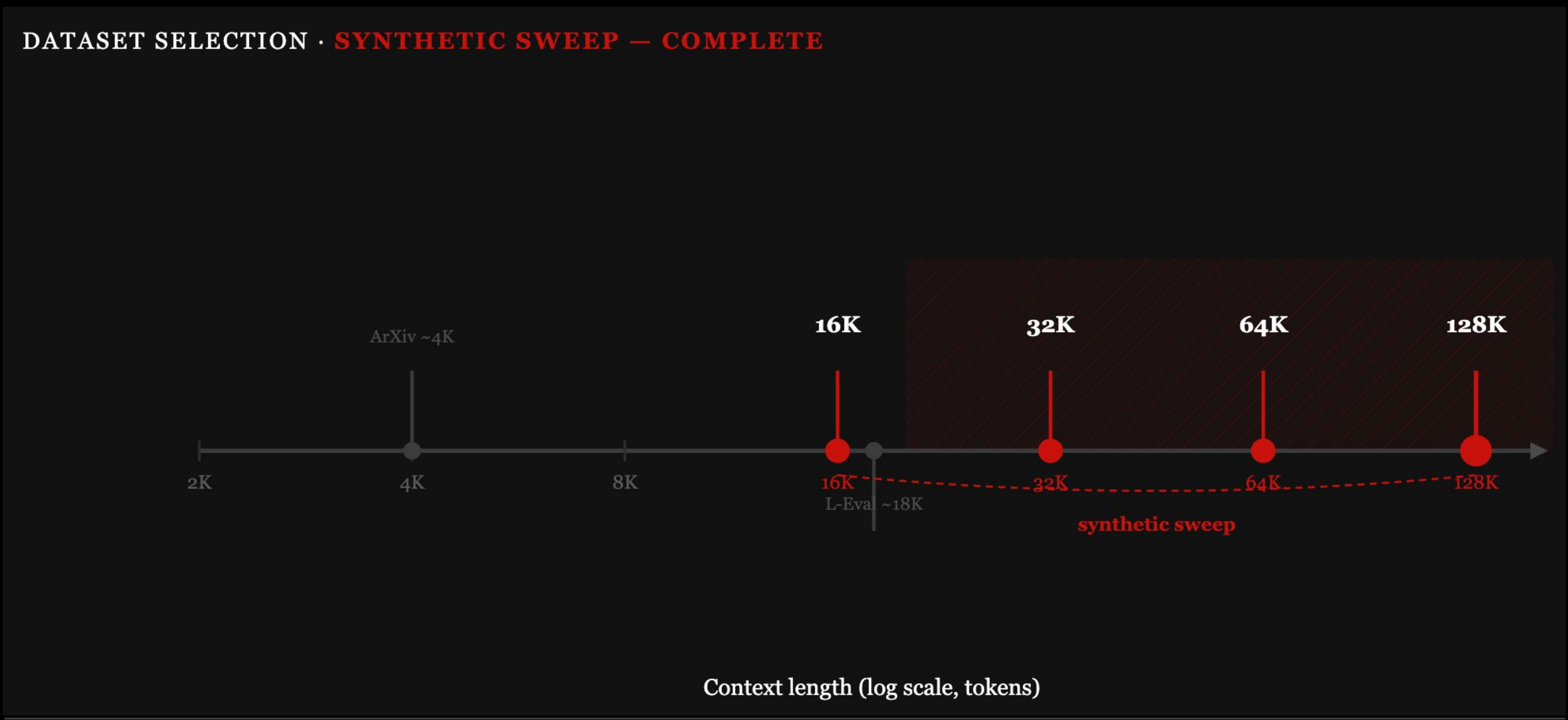
Experiment 1: Results

Experiment 2: Simulated Data

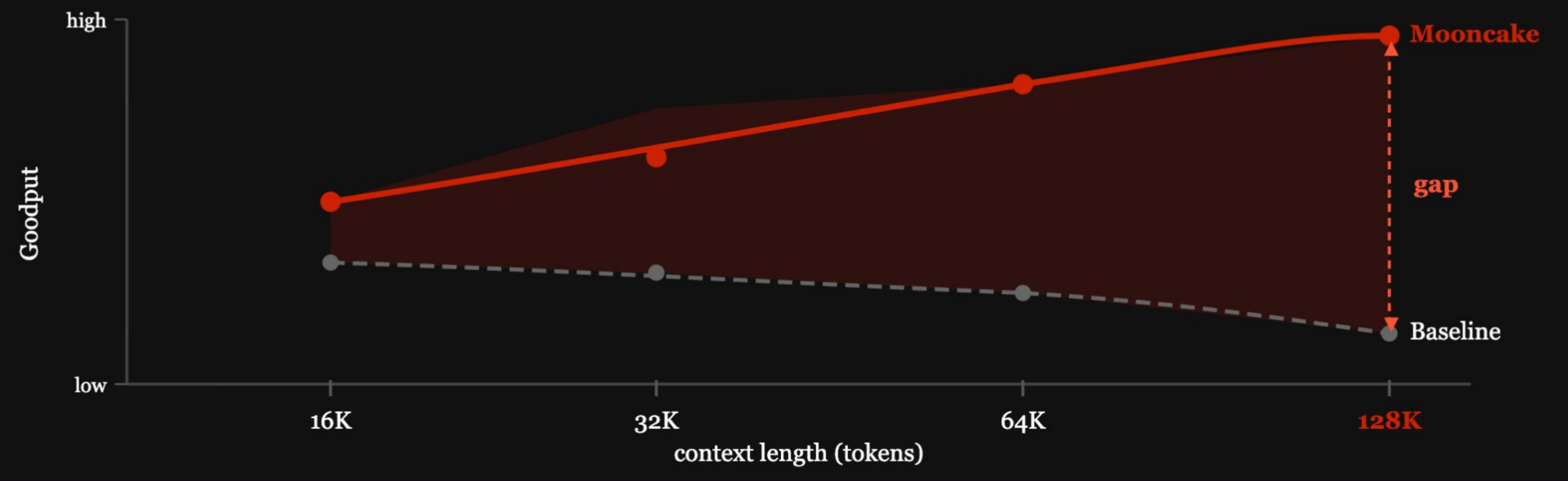
Experiment 2: Simulated Data



Experiment 2: Simulated Data

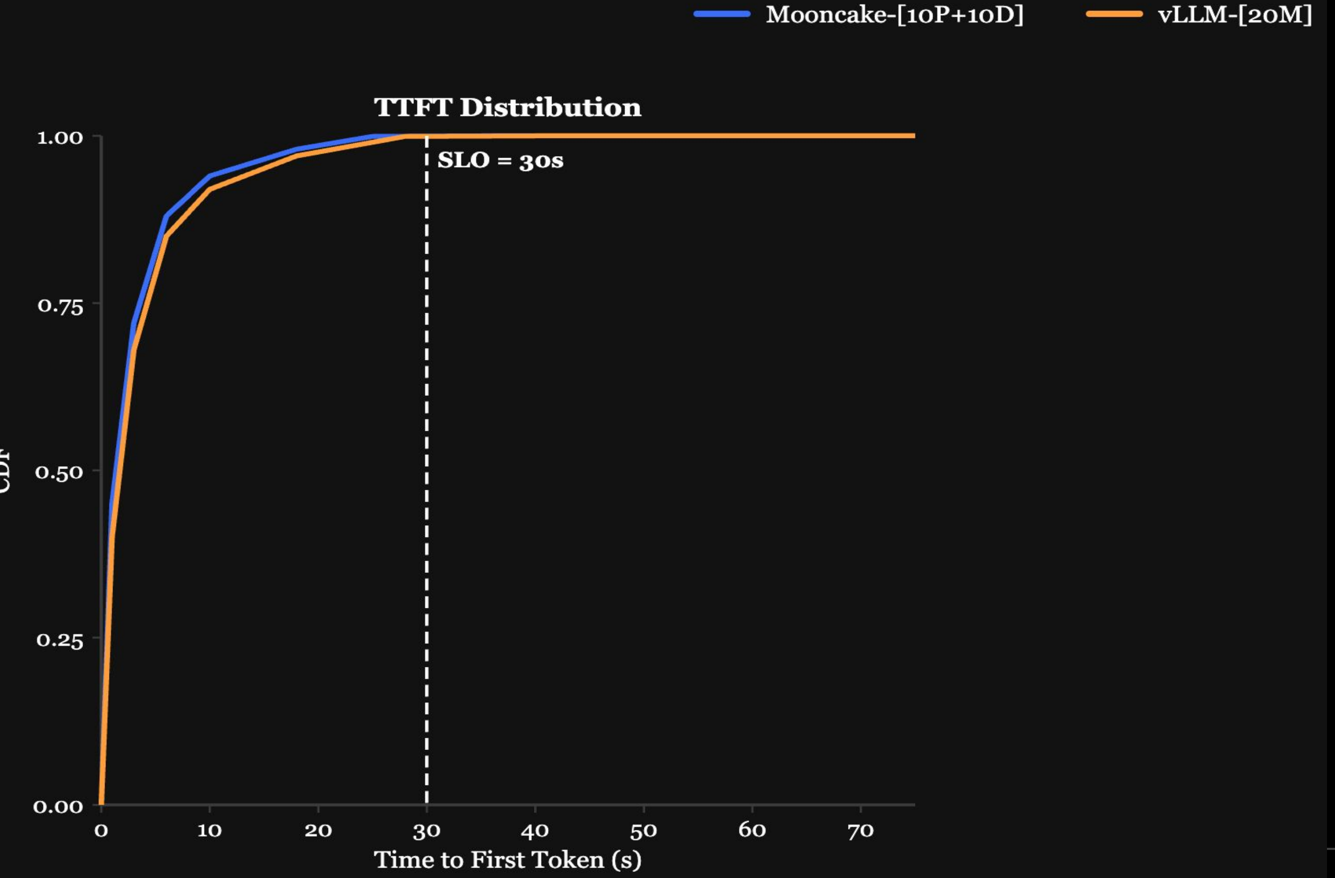


Experiment 2: Results



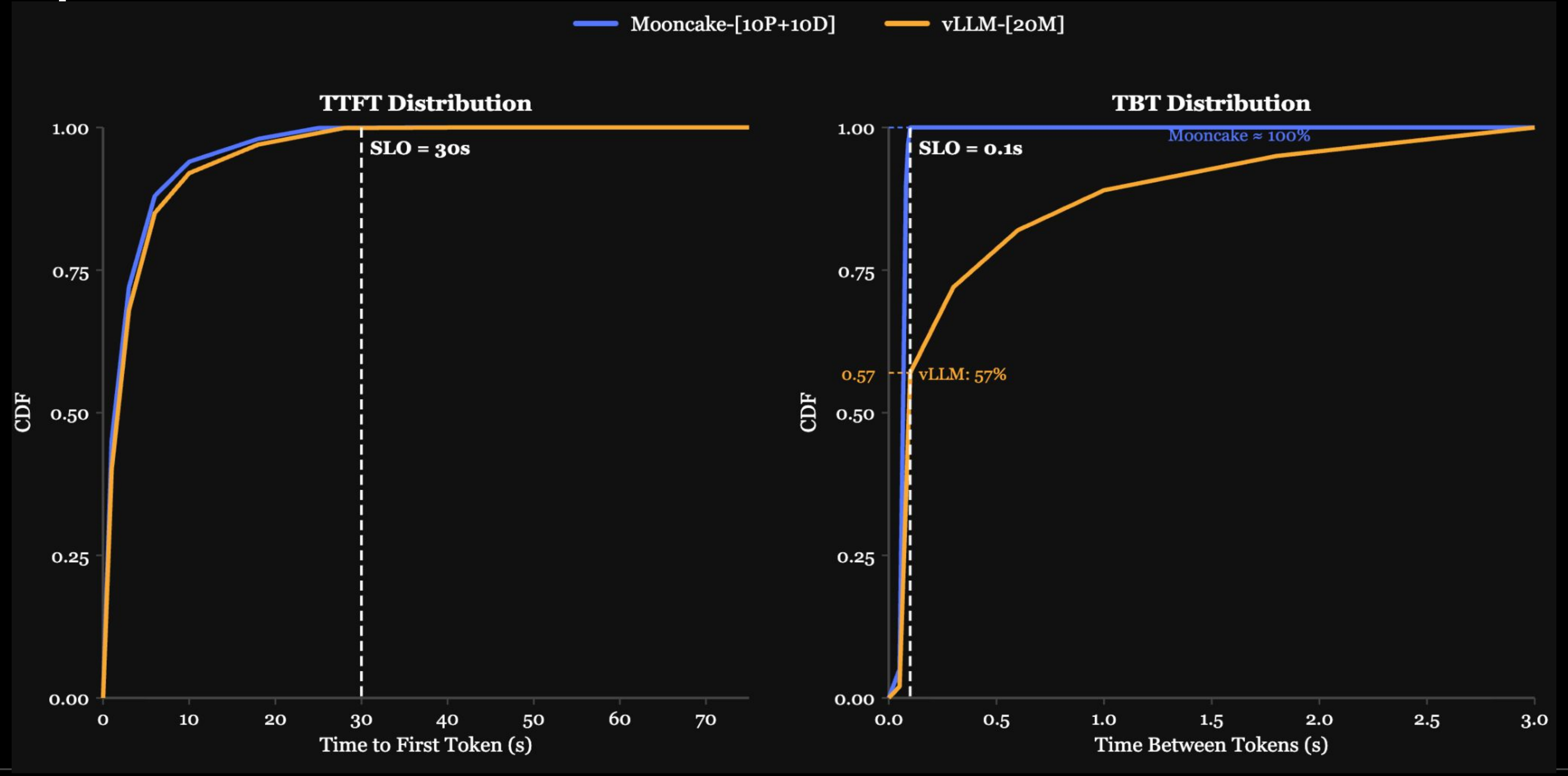
Experiment 3: Real Workload

Experiment 3: Results

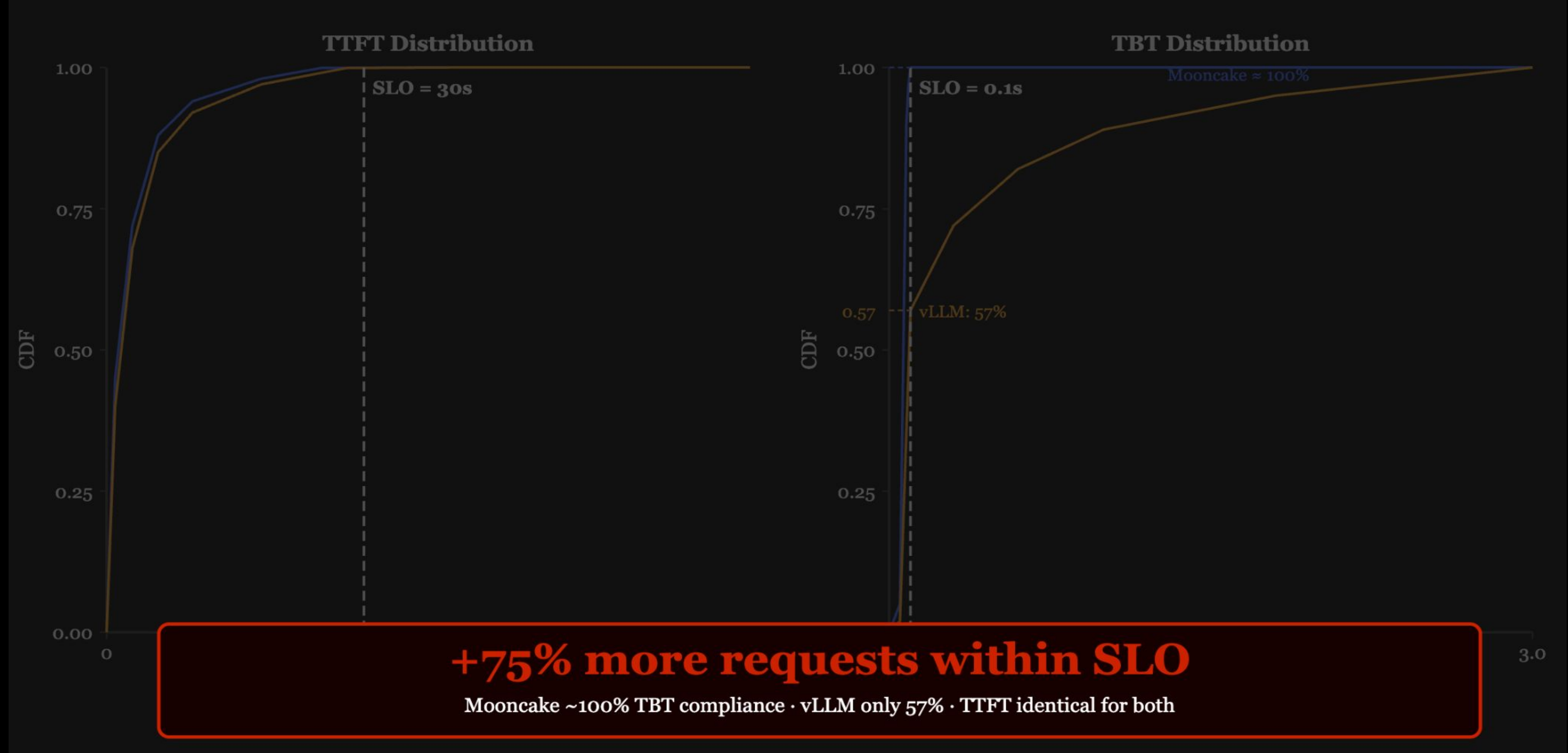


•

Experiment 3: Results



Experiment 3: Results



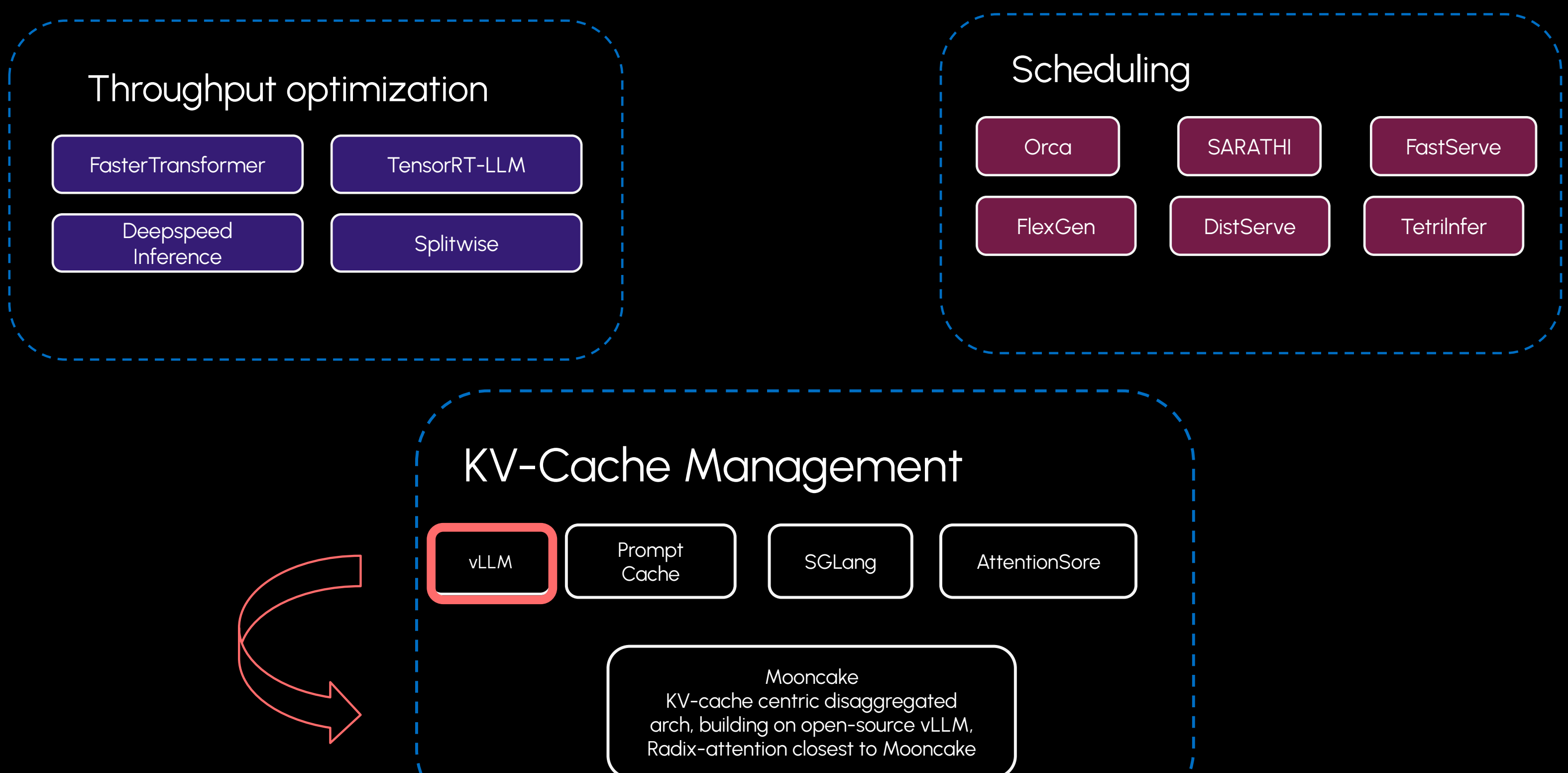
Performance in Overload Scenarios

Rejection Strategy Comparison

Fewer rejections = more requests served = better goodput

Strategy	What it does	Rejections	vs Baseline
Baseline	Reject on load, checks each stage independently — can waste prefill compute	4,183	—
Early Rejection	Check both gates upfront before prefill starts	3,771	−9.8%
Early Rejection + Prediction	Check both gates upfront, using predicted future decode load	3,589	−14.2%

Related Work



Future work