

# Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving



CS2650

Presented by

Raima Afra

Tom Tan

Aadity Sharma

Andrew Zhao

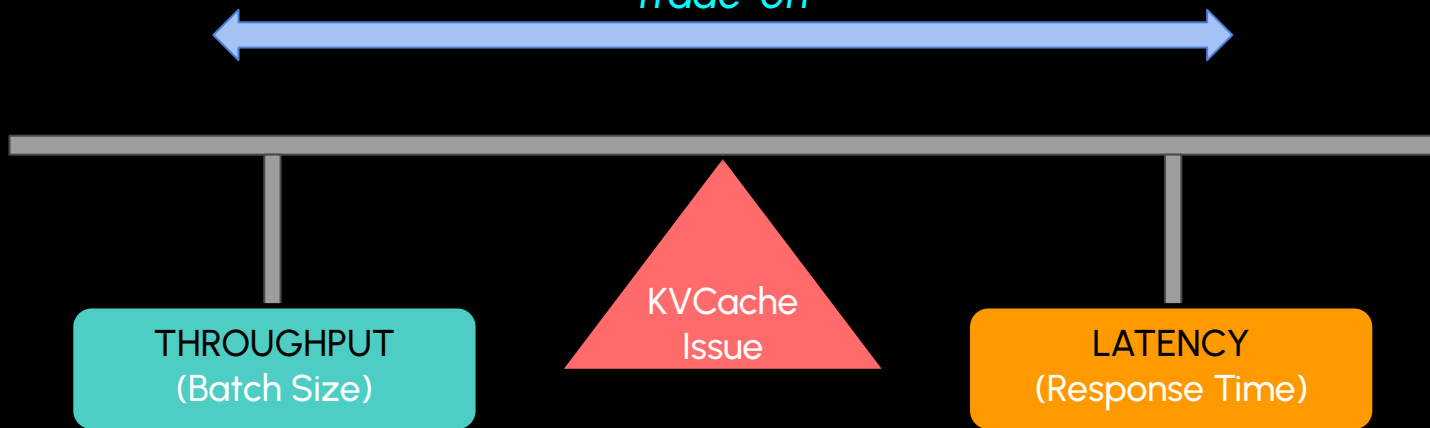
# MOTIVATION

**LLM serving at scale is a resource scheduling problem**

# Throughput vs Latency Trade-off in LLM Inference

*KVCache as the Central Bottleneck*

*Trade-off*



- 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)



TTFT < 300ms

Users perceive this delay immediately

# Key Metric #2: TBT (Time Between Tokens)

GOOD - Smooth Streaming



BAD - Stuttering Output



Target: Each gap < 100ms

# 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



SINGLE GPU

PREFILL: 100k tokens processing

Decoding requests waiting:



VRAM: 80 GB (only 1-2 contexts) | Low cache hit rate

# Mooncake's Core Insight

Same 10k token system prompt used 100 times

## ✗ TRADITIONAL WAY

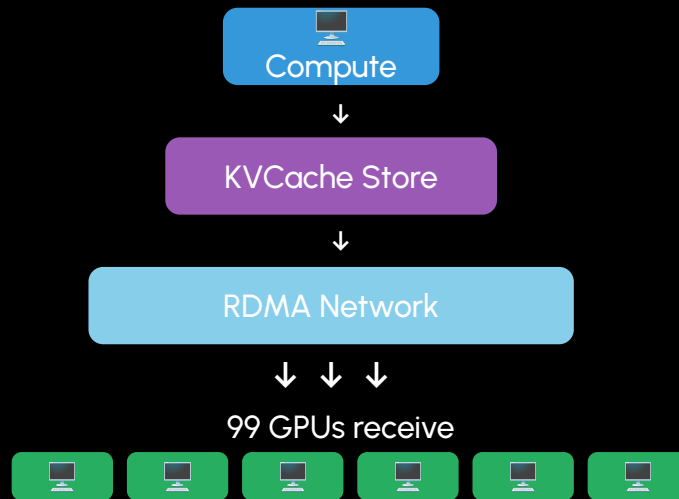
All 100 GPUs computing



Total Cost:

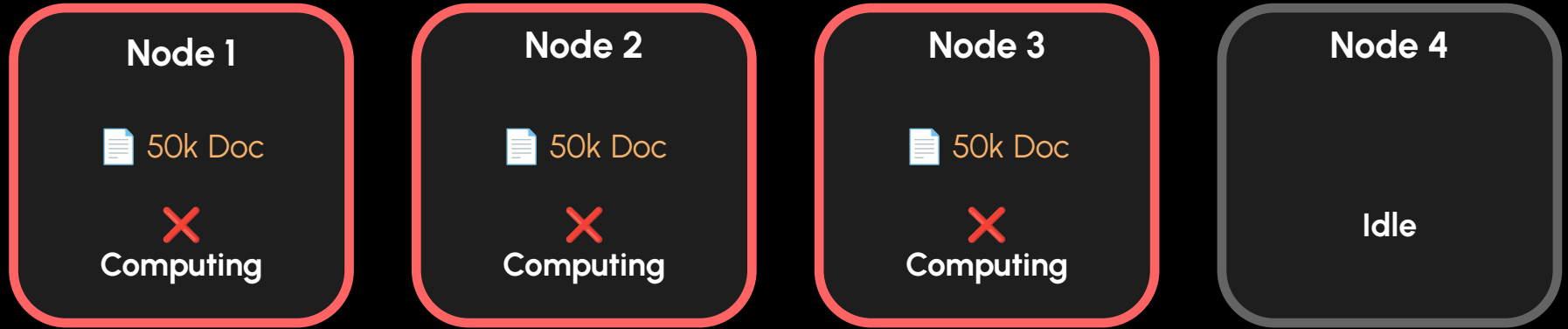
**!! 100 × GPU Computation !!**

## ✓ MOONCAKE WAY



Transfer is almost **50×** CHEAPER!

# Multi-node Systems Waste Computation Without Coordination

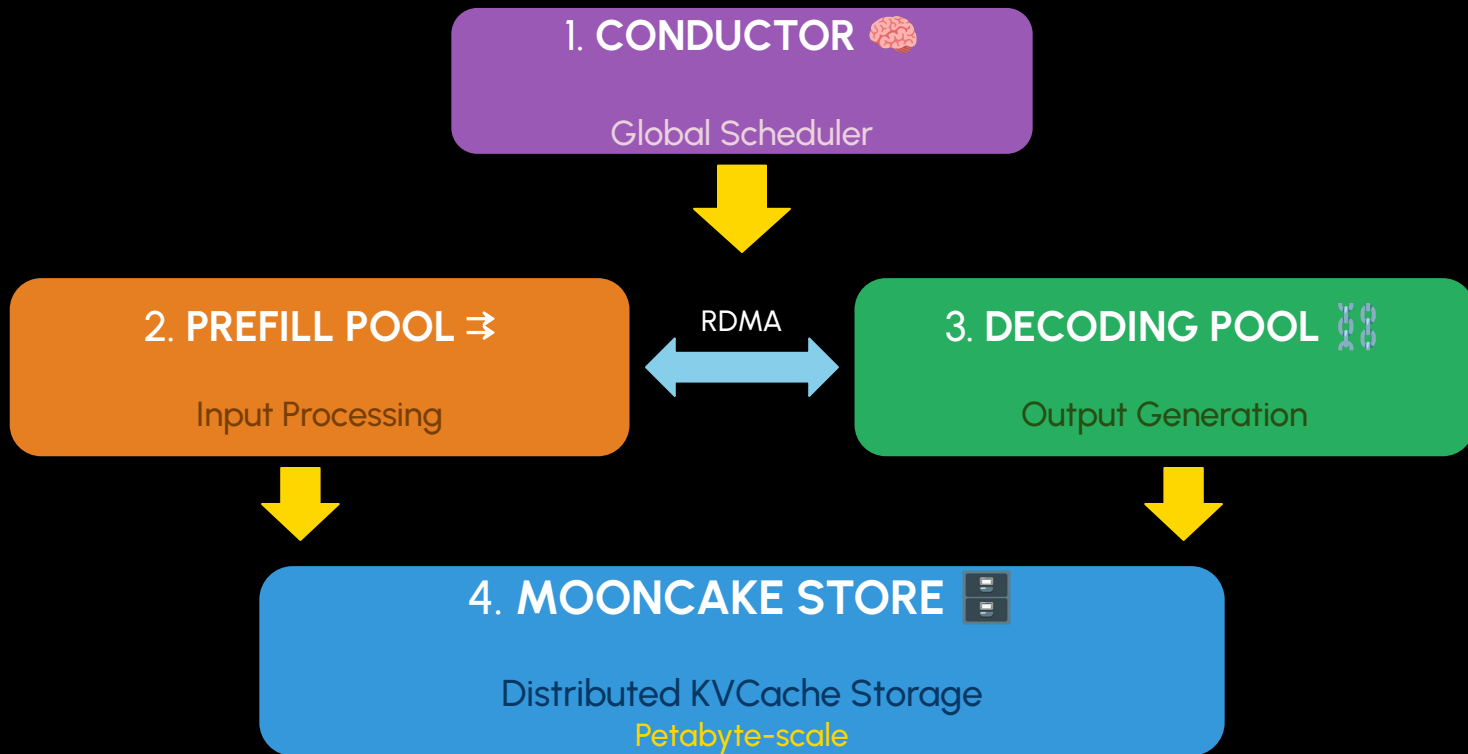


Same 50k token document computed 3 times!

**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

## System Prompt Reuse

 100 Users



Same 10k prompt



Compute once

Reuse 100x

**Savings:**  
99% cache hit

## Multi-turn Chat

 Conversation



Conversation History



Reuse Turns 1-4 & New Question

Compute only new

**Savings:**  
80% cache hit

## Document Q&A

 50k Document



Multiple users



Cache once

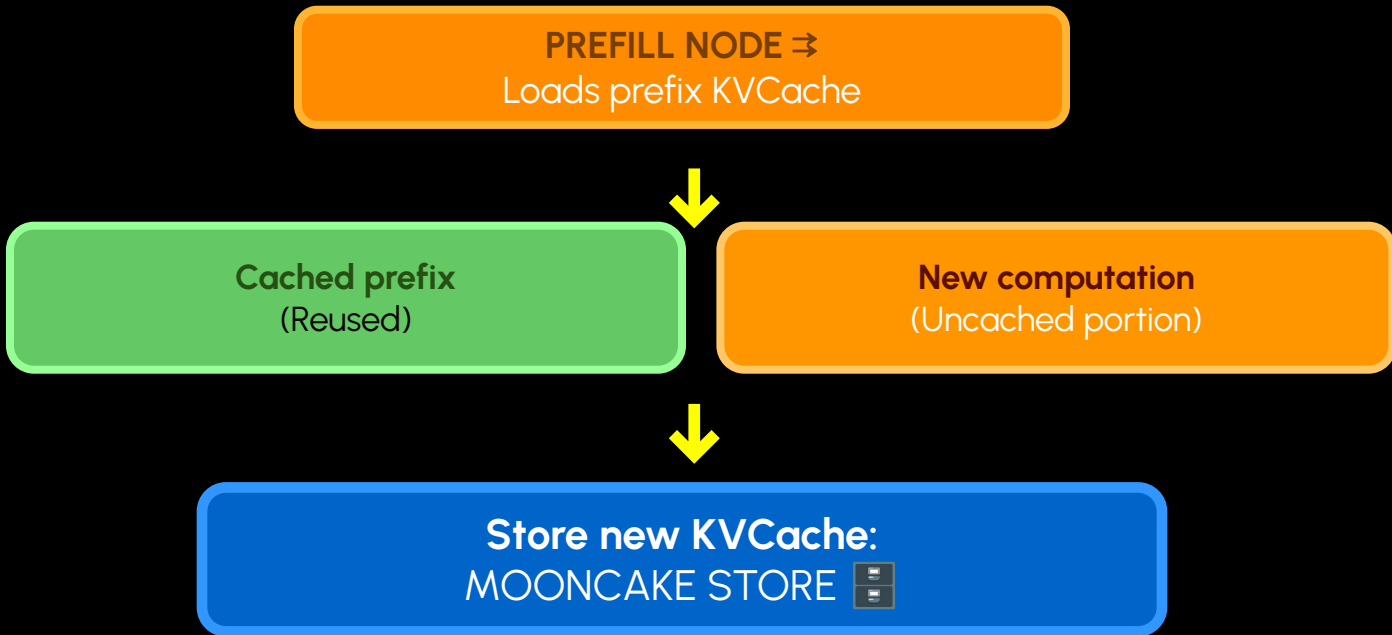
Share all users

**Savings:**  
Massive compute savings

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

# Step 2: Incremental Prefill

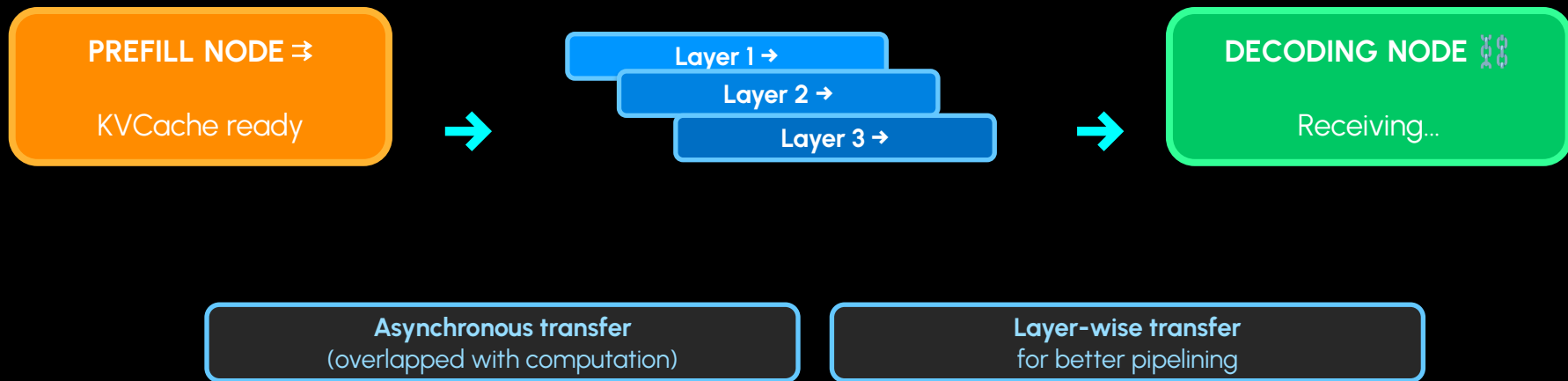
Compute **ONLY** uncached portion



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

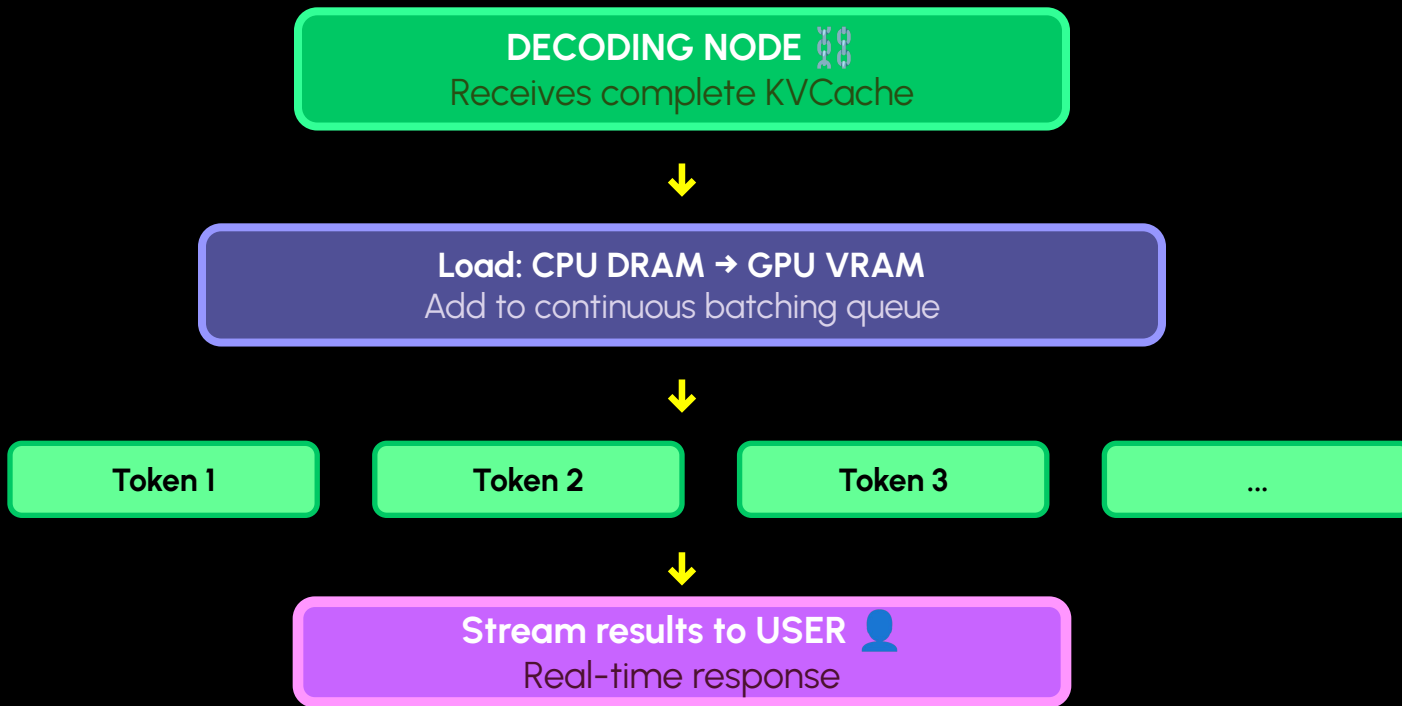
# Step 3: KVCache Transfer

Stream KVCache from Prefill → Decoding

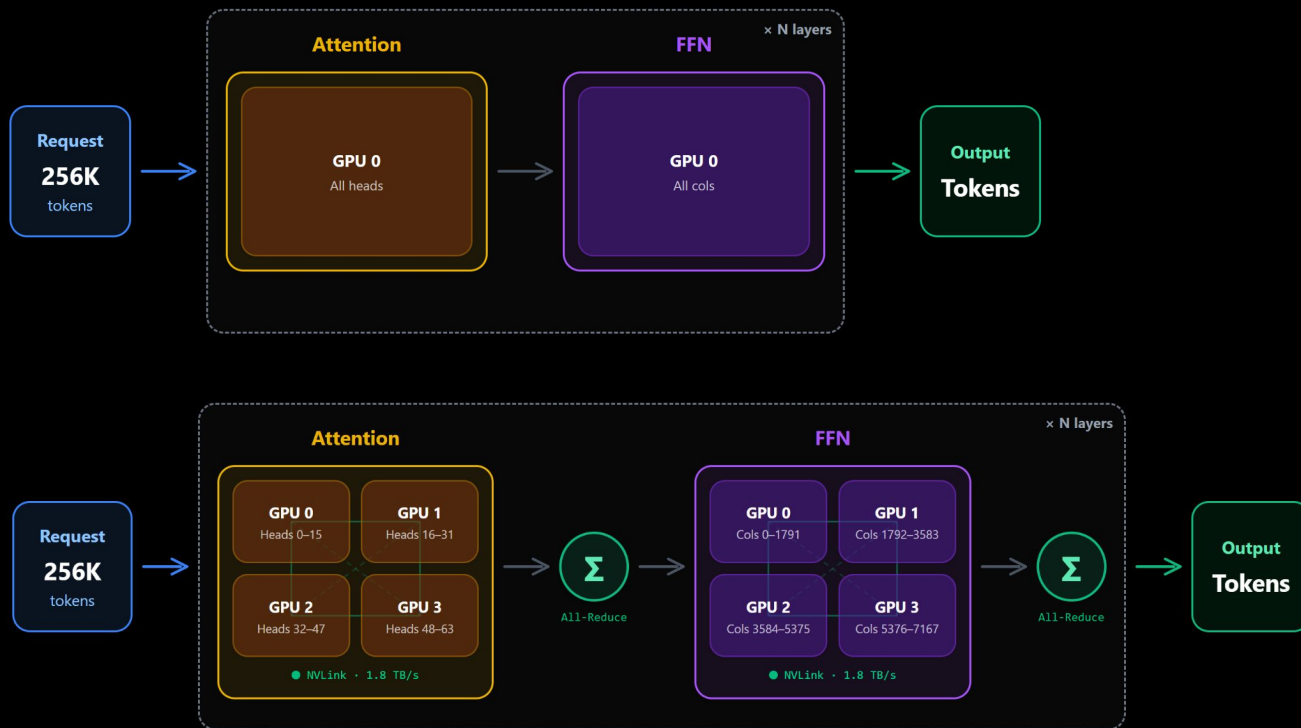


# Step 4: Decoding

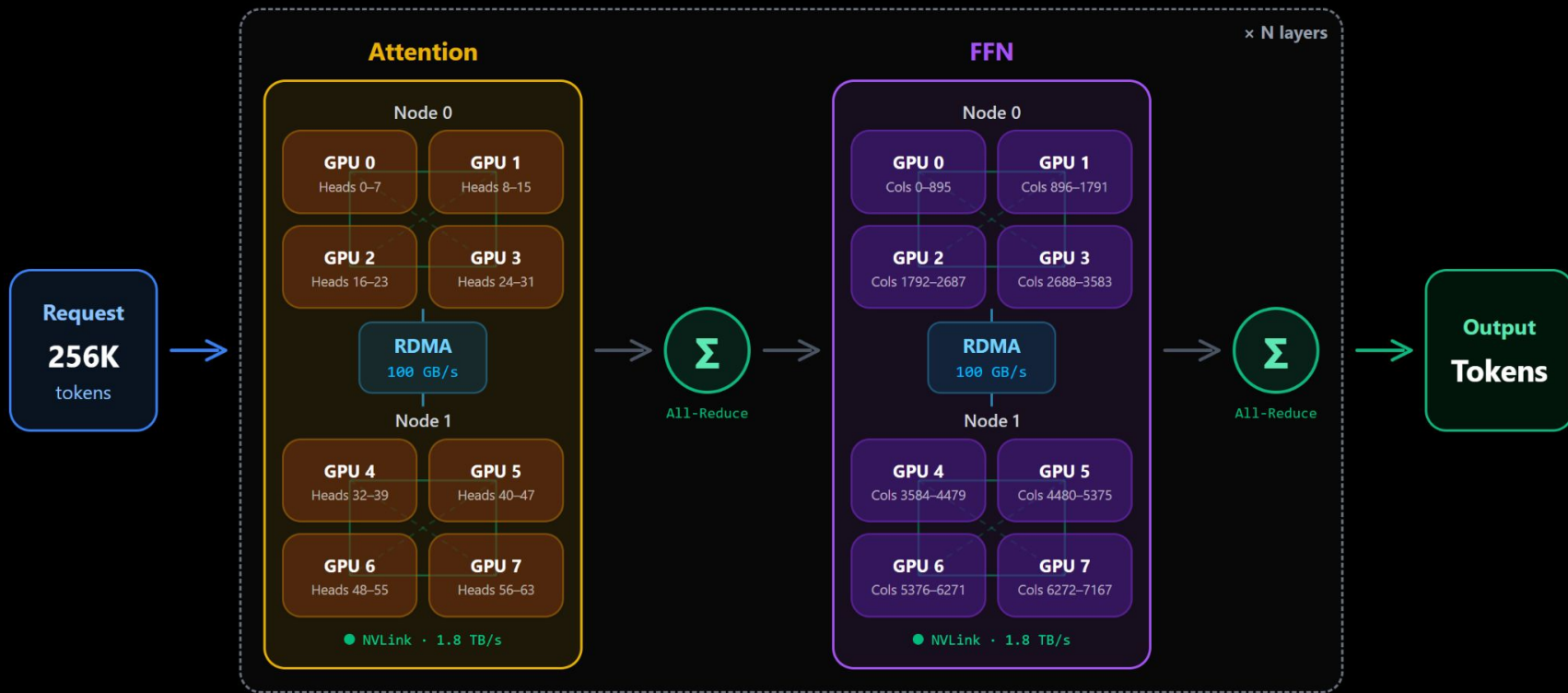
Generate tokens one-by-one and stream to user



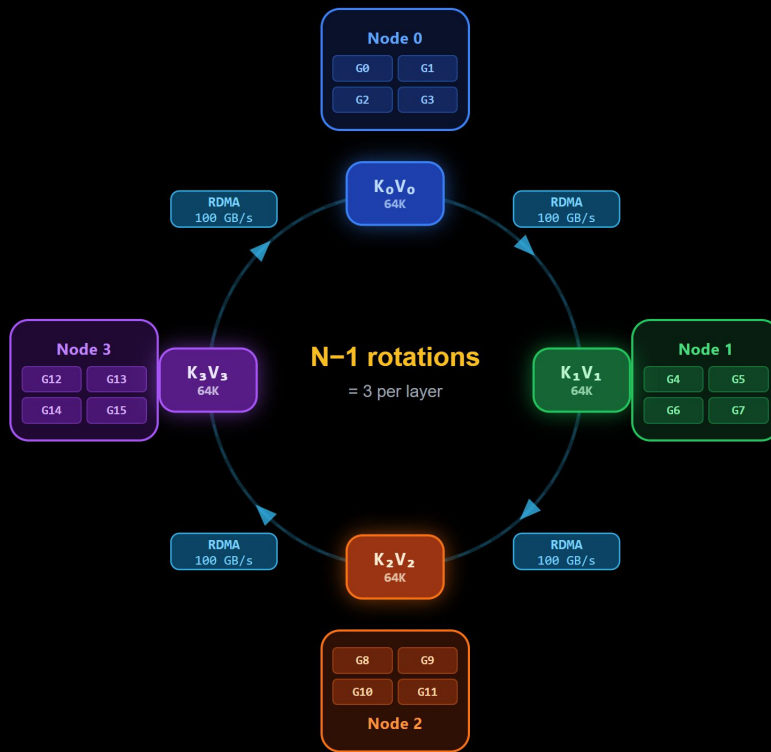
# Scaling up long-context inference



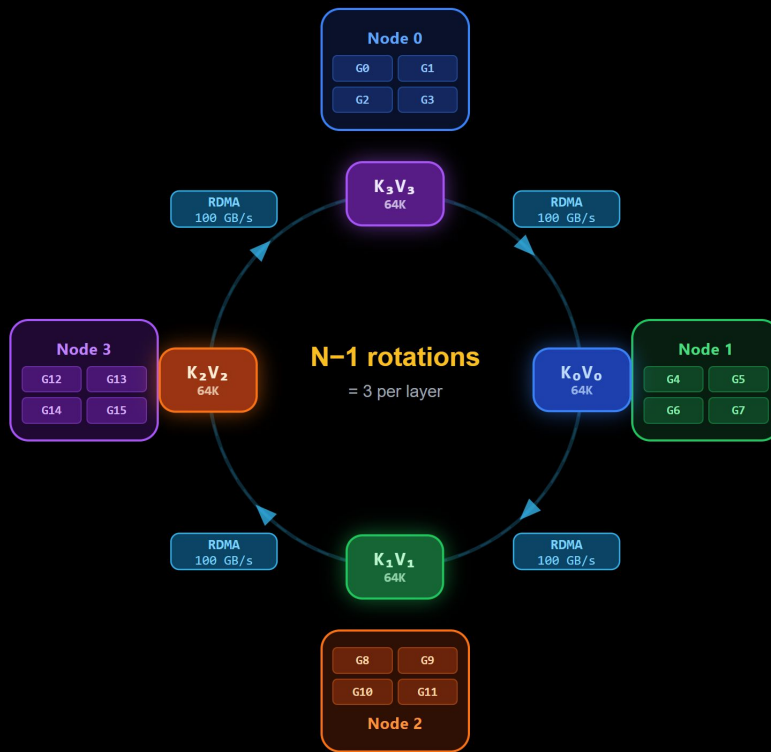
# Scaling beyond single-node



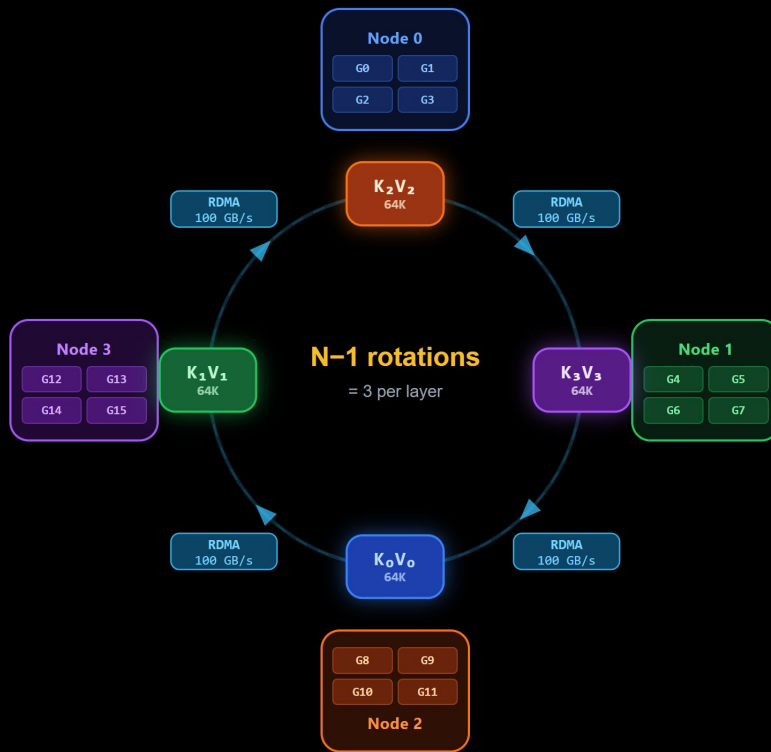
# Existing solution: Sequence parallelism



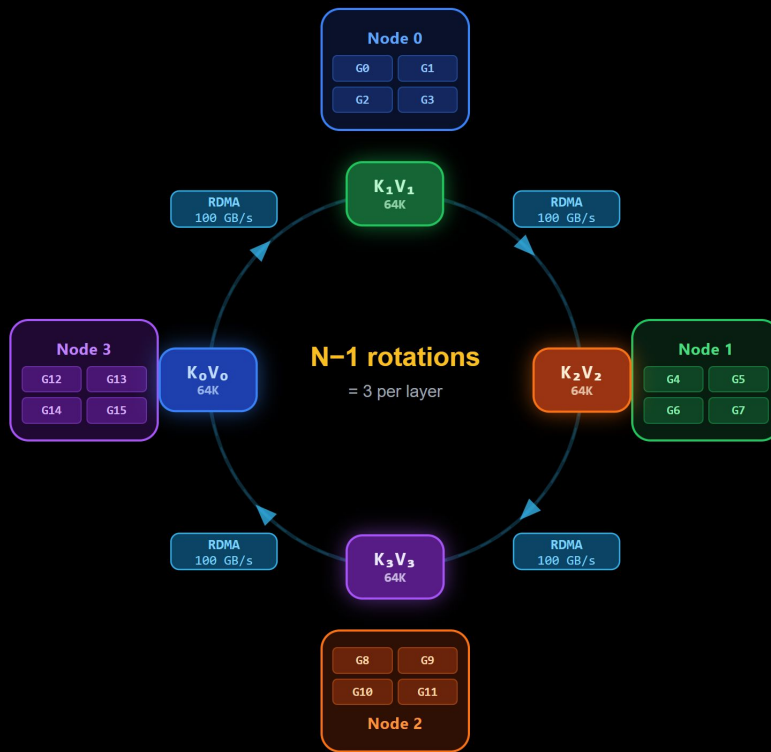
# Existing solution: Sequence parallelism



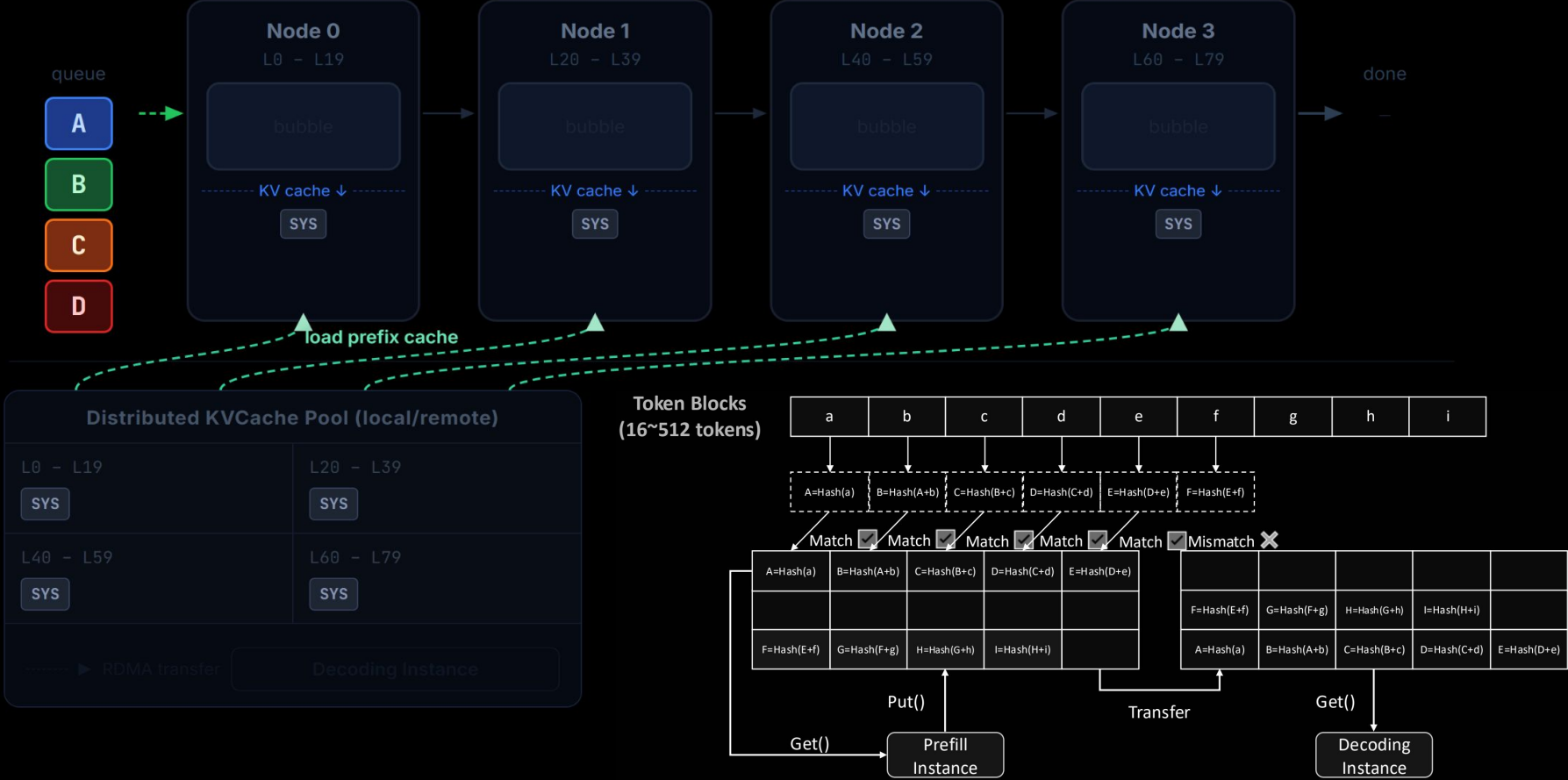
# Existing solution: Sequence parallelism



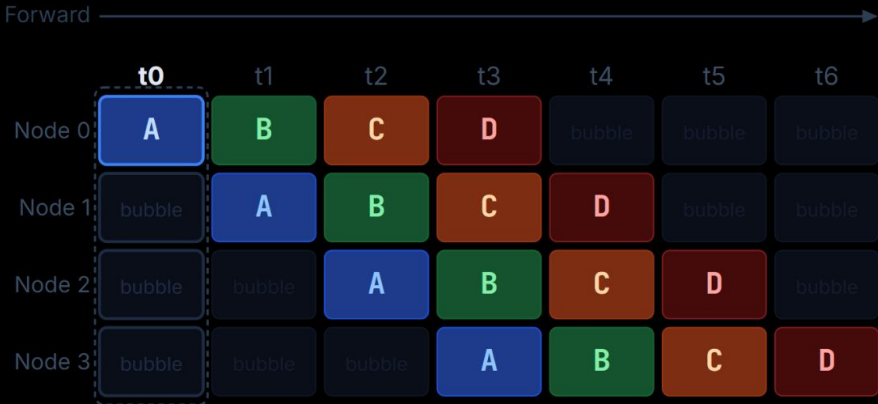
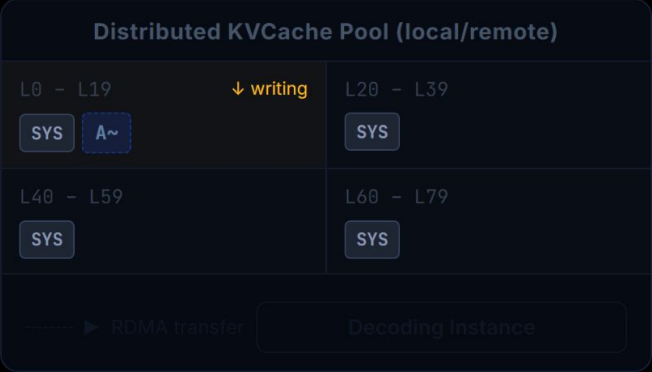
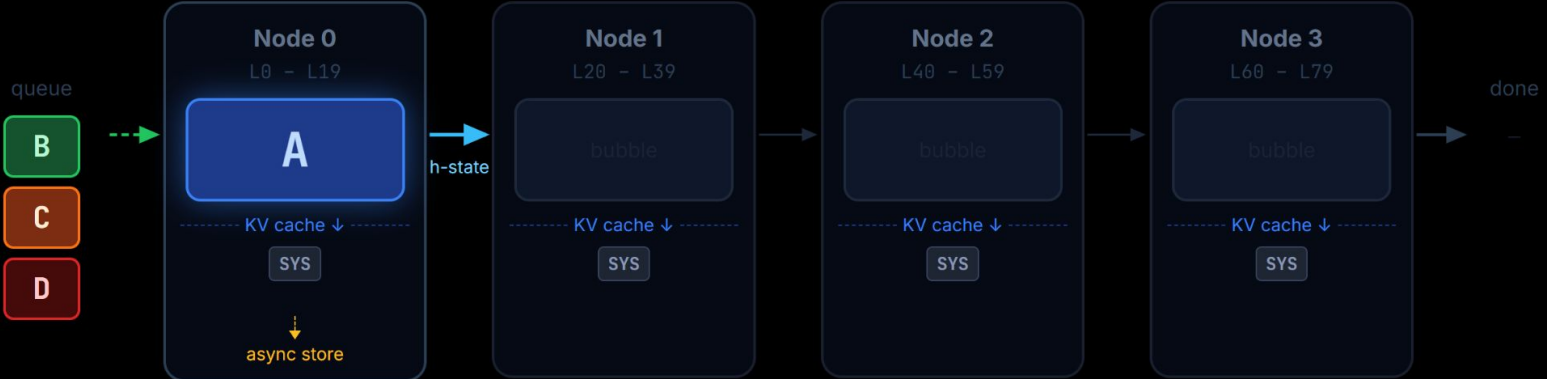
# Existing solution: Sequence parallelism



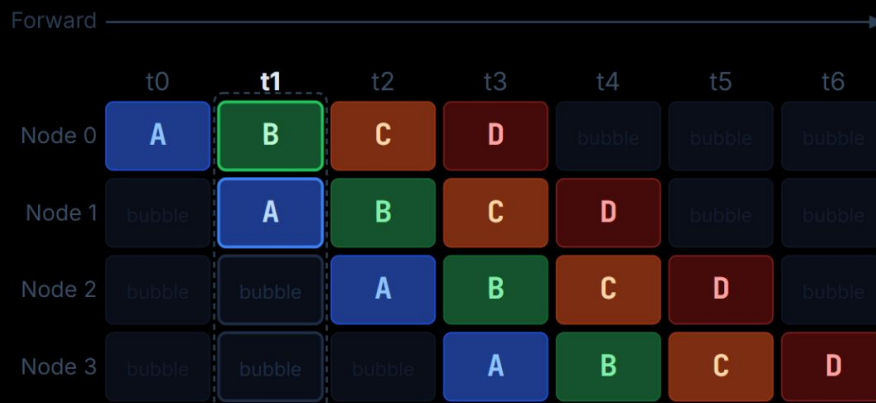
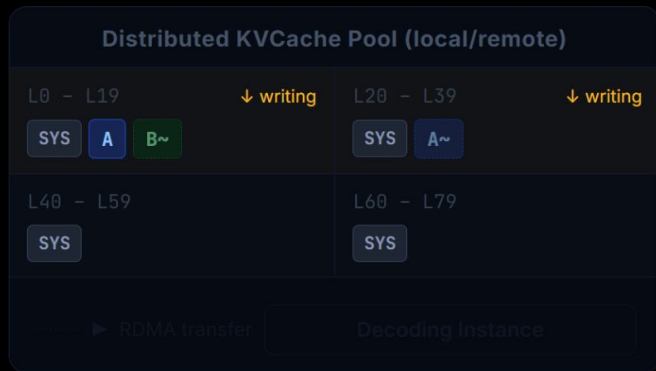
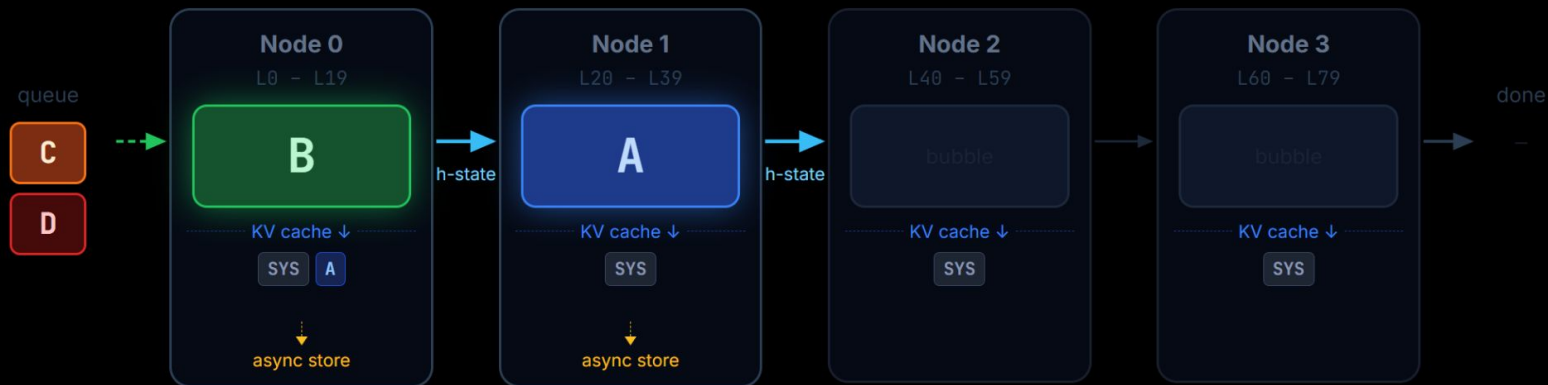
# Chunked Pipeline Parallelism



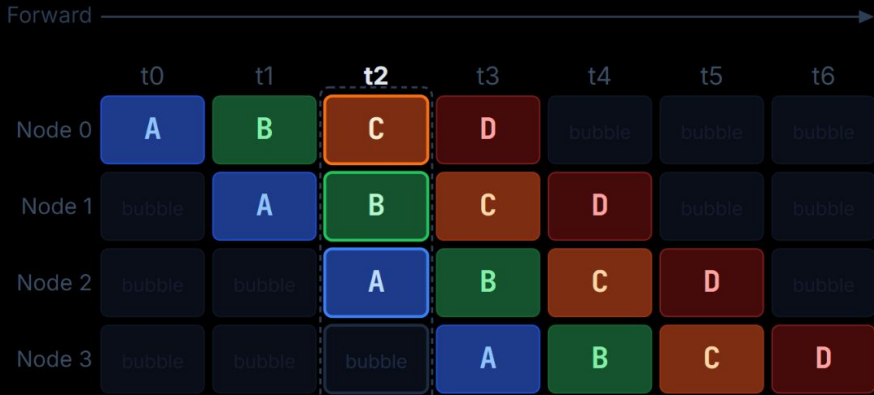
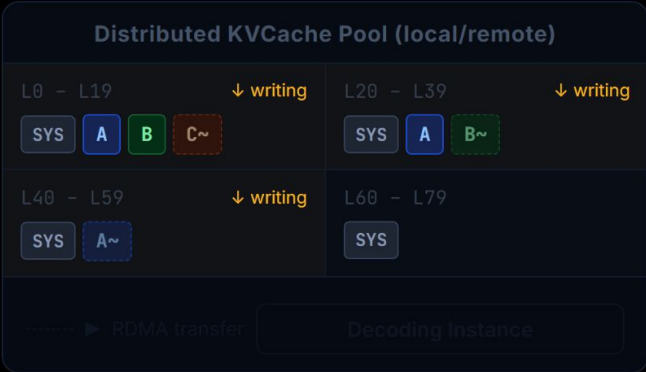
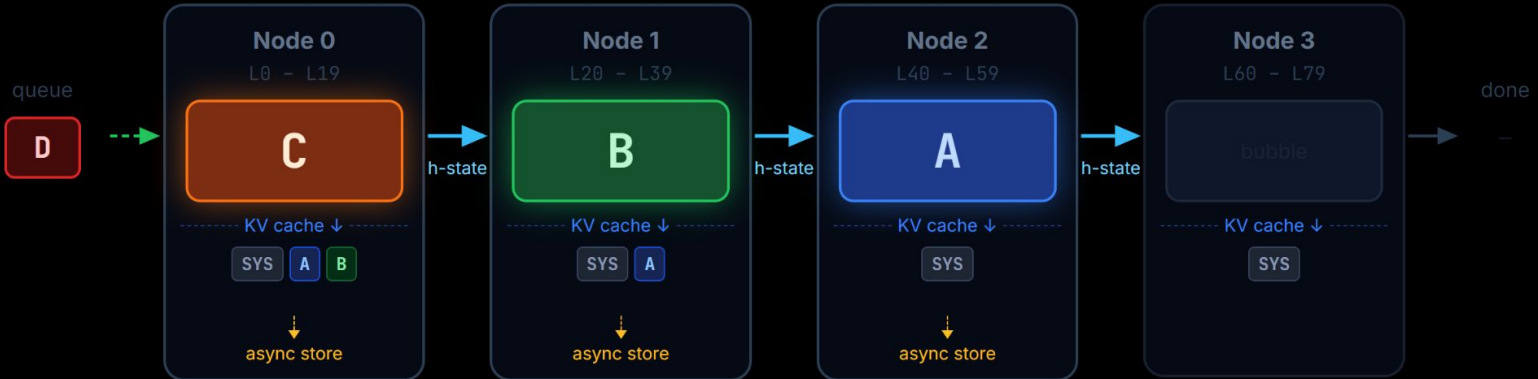
# Chunked Pipeline Parallelism



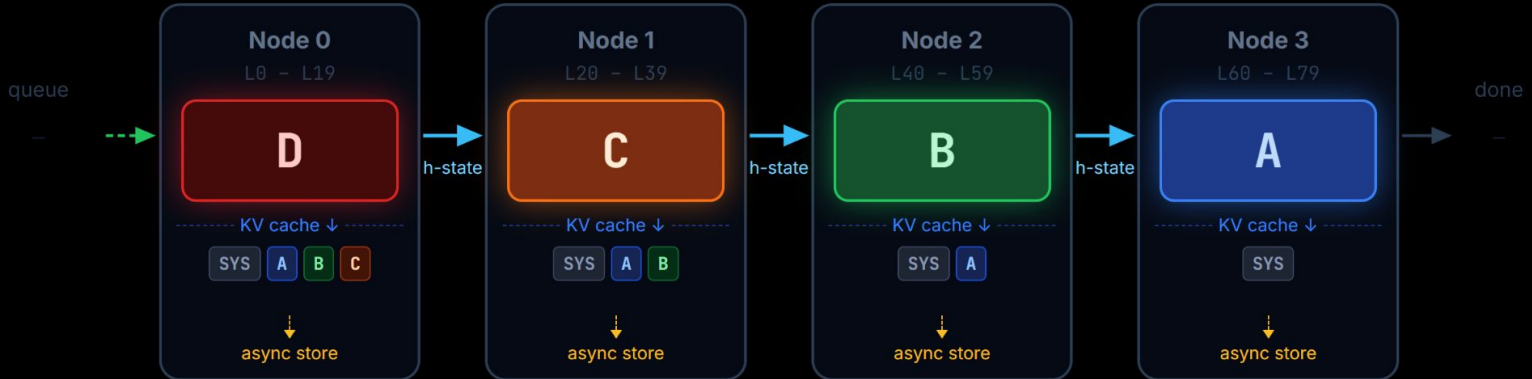
# Chunked Pipeline Parallelism



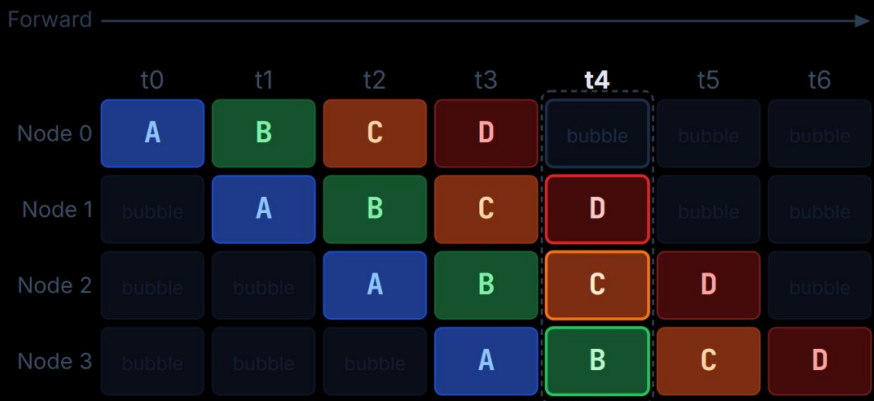
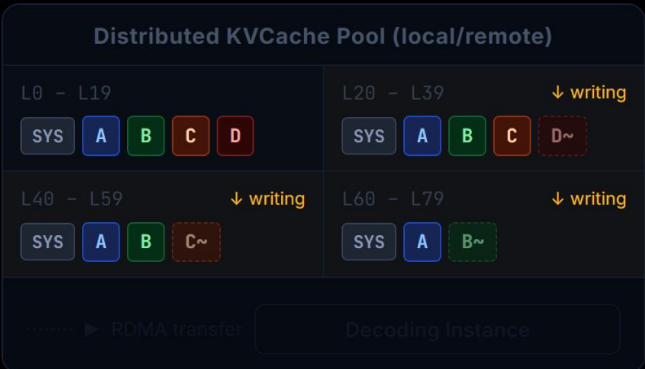
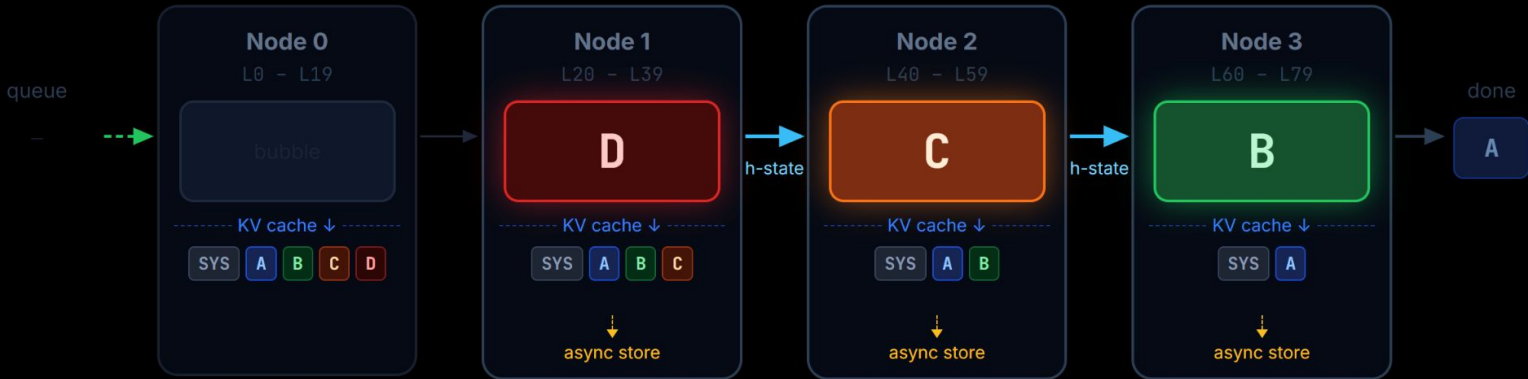
# Chunked Pipeline Parallelism



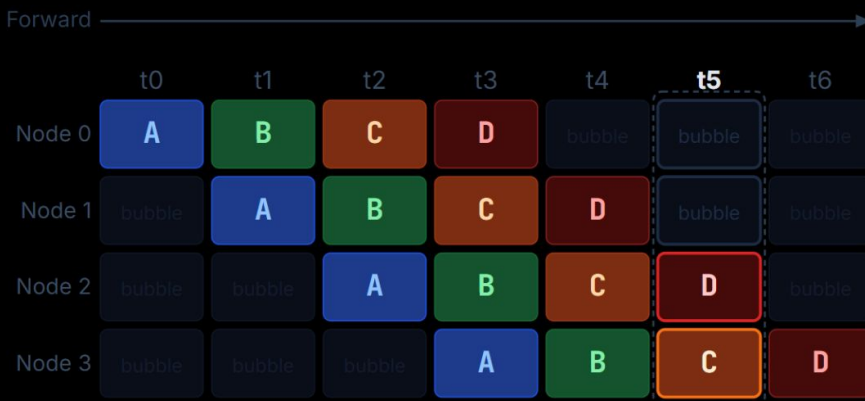
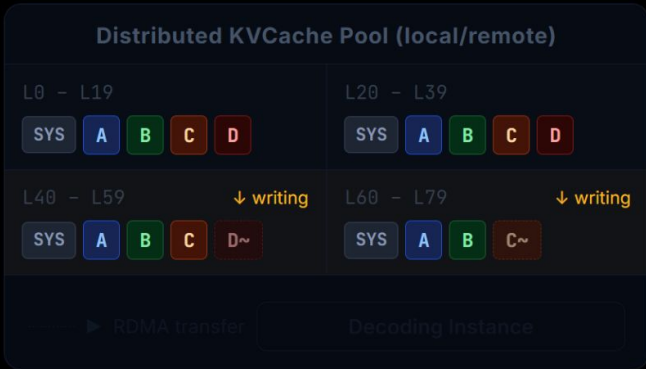
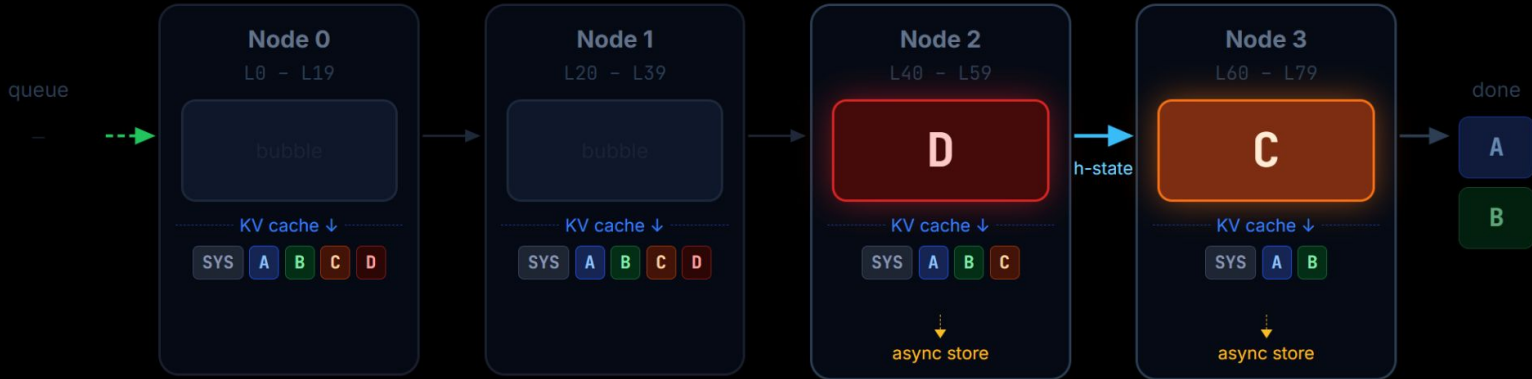
# Chunked Pipeline Parallelism



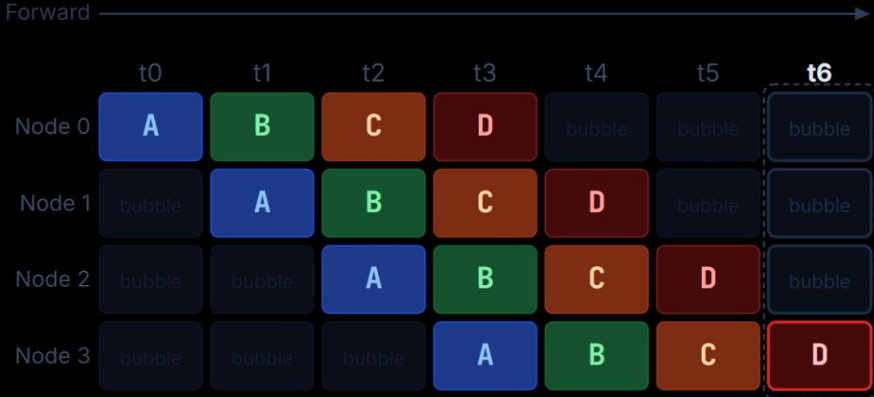
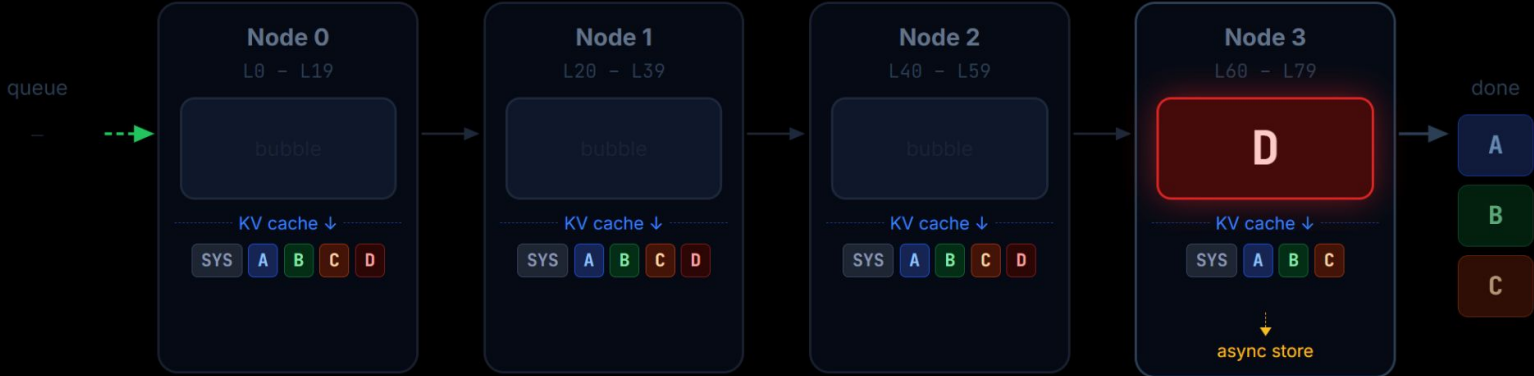
# Chunked Pipeline Parallelism



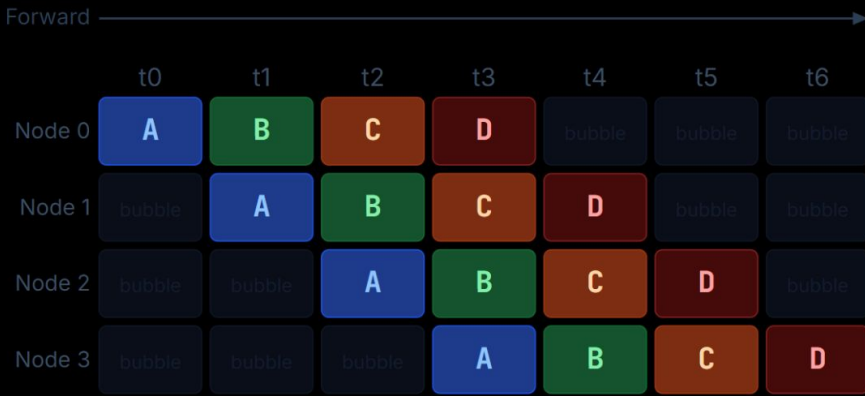
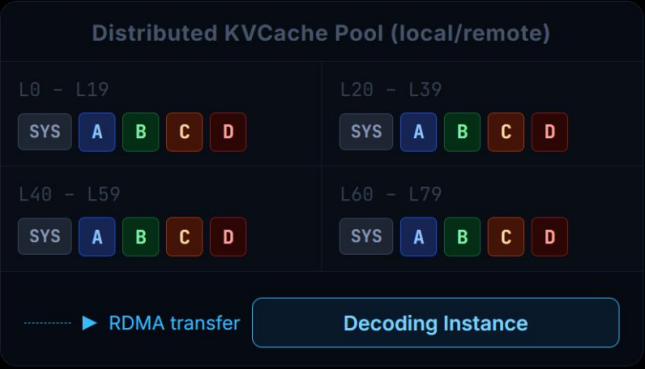
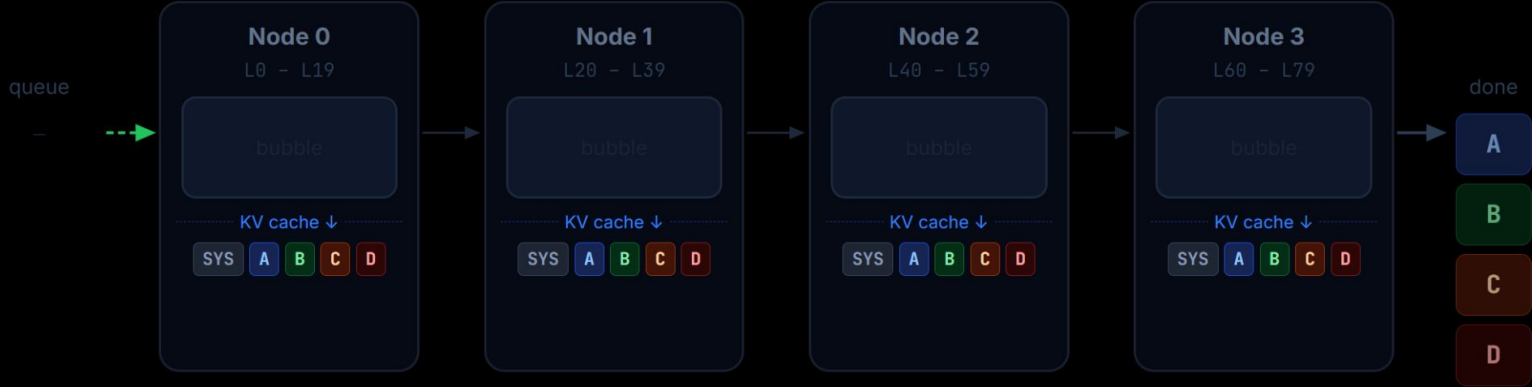
# Chunked Pipeline Parallelism



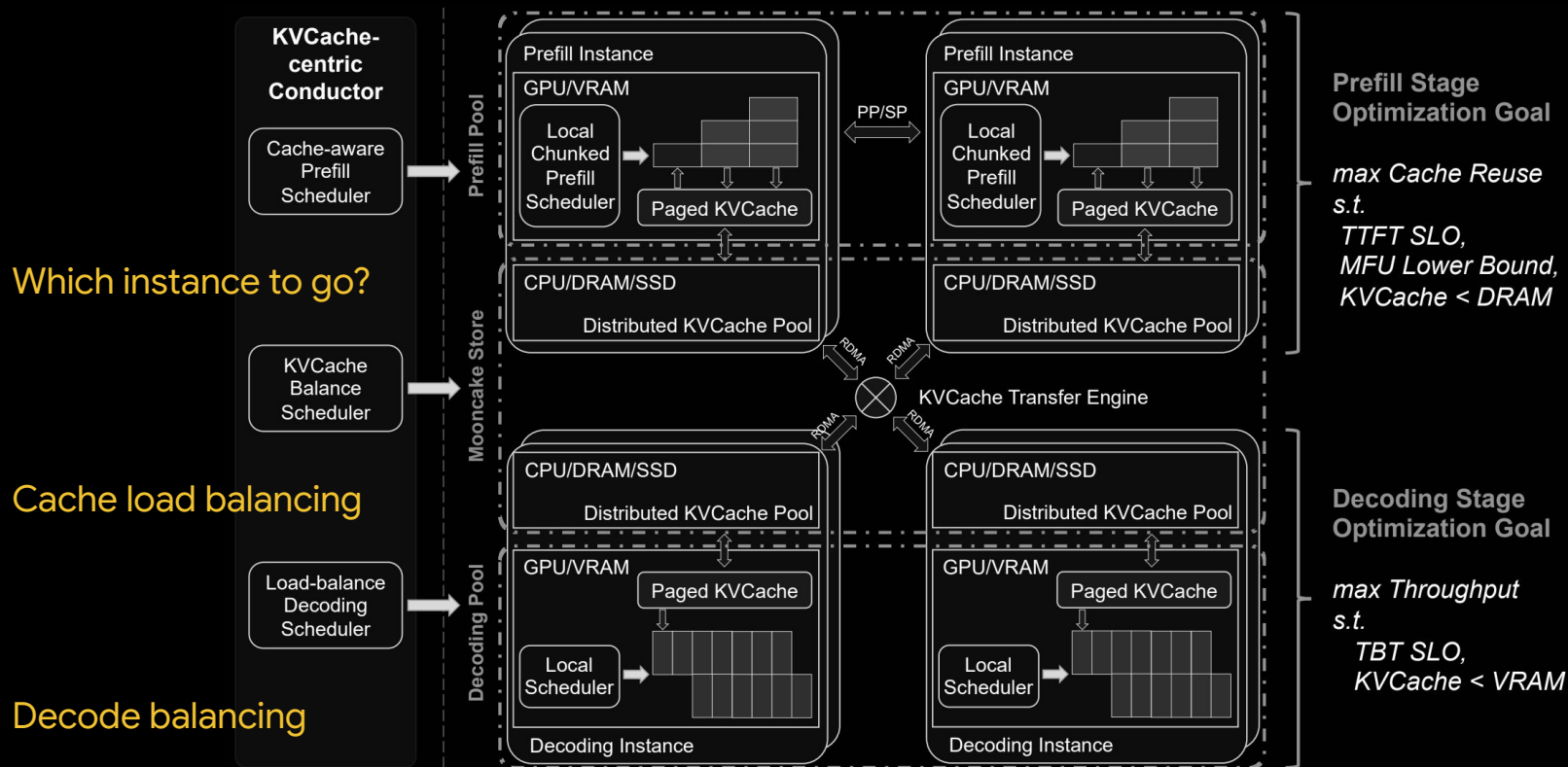
# Chunked Pipeline Parallelism



# Chunked Pipeline Parallelism

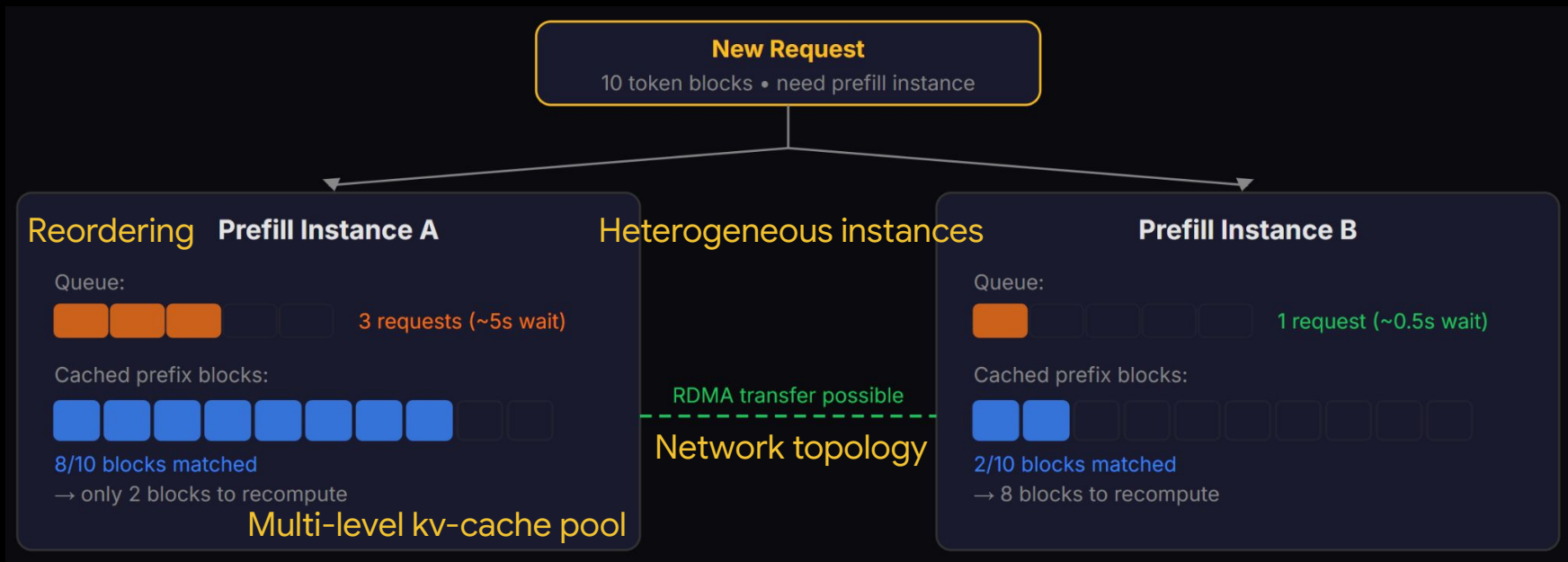


# Mooncake Architecture Overview



# The Scheduling Tension

Addition factor to consider?



## Instance A (local cache)

$T_{\text{queue}} = 5.0\text{s}$  (3 requests ahead)

$T_{\text{prefill}} = 1.0\text{s}$  (2 blocks)

TTFT = 6.0s

## Instance B (local cache)

$T_{\text{queue}} = 0.5\text{s}$  (1 requests ahead)

$T_{\text{prefill}} = 4.0\text{s}$  (8 blocks)

TTFT = 4.5s

## Instance B (RDMA transfer)

$T_{\text{transfer}} = 2.0\text{s}$  (RDMA, 6 block)

$T_{\text{queue}} + T_{\text{prefill}} = 1.5\text{s}$

TTFT = 3.5s

# The Scheduler Design

## Prefix hashing and matching

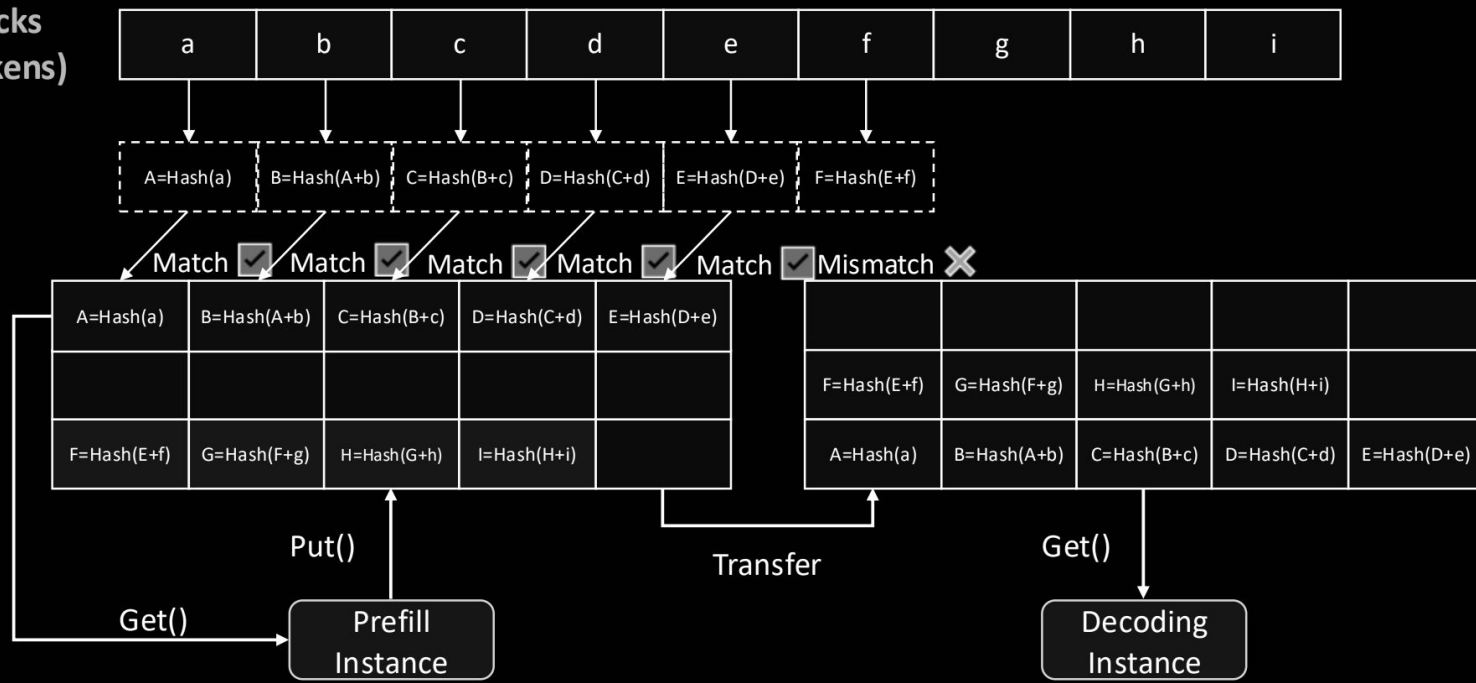
### Algorithm 1 KVCache-centric Scheduling Algorithm

**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:  $TFTT \leftarrow \text{inf}$
- 3:  $p \leftarrow \emptyset$
- 4:  $best\_prefix\_len, best\_matched\_instance \leftarrow \text{FindBestPrefixMatch}(P, block\_keys)$
- 5: **for**  $instance \in P$  **do**

Token Blocks  
(16~512 tokens)



# The Scheduler Design

Prefix hashing and matching

Compute in Parallel

Strategy A (local cache)

Manually tuned

$$\frac{\text{best\_prefix\_len}}{\text{prefix\_len}} < \text{kvcache\_balancing\_threshold}$$

Prediction model to estimate this time

## 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 ← PrefixHash( $R.prompt\_tokens$ ,  $B$ )
2: TTFT ← inf
3:  $p \leftarrow \emptyset$ 
4: best_prefix_len, best_matched_instance ← FindBestPrefixMatch( $P$ , block_keys)
5: for instance ∈  $P$  do
6:   prefix_len ← instance.prefix_len
7:   T_queue ← EstimatePrefillQueueTime(instance)
8:   if  $\frac{\text{best\_prefix\_len}}{\text{prefix\_len}} < \text{kvcache\_balancing\_threshold}$  then
9:     T_prefill ← EstimatePrefillExecutionTime(len( $R.prompt\_tokens$ ), prefix_len)
10:    if TTFT > T_queue + T_prefill then
11:      TTFT ← T_queue + T_prefill
12:       $p \leftarrow \text{instance}$ 
13:    end if
14:  else
15:    transfer_len ← best_prefix_len - prefix_len
16:    T_transfer ← EstimateKVCacheTransferTime(instance, best_matched_instance, transfer_len)
17:    T_prefill ← EstimatePrefillExecutionTime(len( $R.prompt\_tokens$ ), best_prefix_len)
18:    if TTFT > T_transfer + T_queue + T_prefill then
19:      TTFT ← T_transfer + T_queue + T_prefill
20:       $p \leftarrow \text{instance}$ 
21:    end if
22:  end if
23: end for
24:  $d, TBT \leftarrow \text{SelectDecodingInstance}(D)$ 
25: if TTFT > TTFT_SLO or TBT > TBT_SLO then
26:   reject  $R$ ; return
27: end if
28: if  $\frac{\text{best\_prefix\_len}}{p.prefix\_len} > \text{kvcache\_balancing\_threshold}$  then
29:   TransferKVCache(best_matched_instance,  $p$ )
30: end if
31: return ( $p, d$ )

```

Sum of prefill exec time

SLO target

▷ Load-balancing decoding scheduling

▷ KVCache hot-spot migration

# 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

### 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:  $TFTT \leftarrow \text{inf}$ 
3:  $p \leftarrow \emptyset$ 
4:  $best\_prefix\_len, best\_matched\_instance \leftarrow \text{FindBestPrefixMatch}(P, block\_keys)$ 
5: for  $instance \in P$  do
6:    $prefix\_len \leftarrow instance.prefix\_len$ 
7:    $T_{queue} \leftarrow \text{EstimatePrefillQueueTime}(instance)$  Sum of prefill exec time
8:   if  $\frac{best\_prefix\_len}{prefix\_len} < kvcache\_balancing\_threshold$  then ▷ Cache-aware prefill scheduling
9:      $T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt\_tokens), prefix\_len)$ 
10:    if  $TFTT > T_{queue} + T_{prefill}$  then SLO target
11:       $TFTT \leftarrow T_{queue} + T_{prefill}$ 
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:     $T_{transfer} \leftarrow \text{EstimateKVCacheTransferTime}(instance, best\_matched\_instance, transfer\_len)$ 
17:     $T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt\_tokens), best\_prefix\_len)$ 
18:    if  $TFTT > T_{transfer} + T_{queue} + T_{prefill}$  then
19:       $TFTT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$ 
20:       $p \leftarrow instance$ 
21:    end if
22:  end if
23: end for
24:  $d, TBT \leftarrow \text{SelectDecodingInstance}(D)$  ▷ Load-balancing decoding scheduling
25: if  $TFTT > TFTT\_SLO$  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:    $\text{TransferKVCache}(best\_matched\_instance, p)$  ▷ 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  $R$ .

```
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:    $prefix\_len \leftarrow instance.prefix\_len$ 
7:    $T_{queue} \leftarrow \text{EstimatePrefillQueueTime}(instance)$ 
8:   if  $\frac{best\_prefix\_len}{prefix\_len} < kvcache\_balancing\_threshold$  then ▷ Cache-aware prefill scheduling
9:      $T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt\_tokens), prefix\_len)$ 
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:     $T_{transfer} \leftarrow \text{EstimateKVCacheTransferTime}(instance, best\_matched\_instance, transfer\_len)$ 
17:     $T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt\_tokens), best\_prefix\_len)$ 
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:  $d, TBT \leftarrow \text{SelectDecodingInstance}(D)$  ▷ 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:    $\text{TransferKVCache}(best\_matched\_instance, p)$  ▷ KVCache hot-spot migration
30: end if
31: return  $(p, d)$ 
```

Prefix hashing and matching

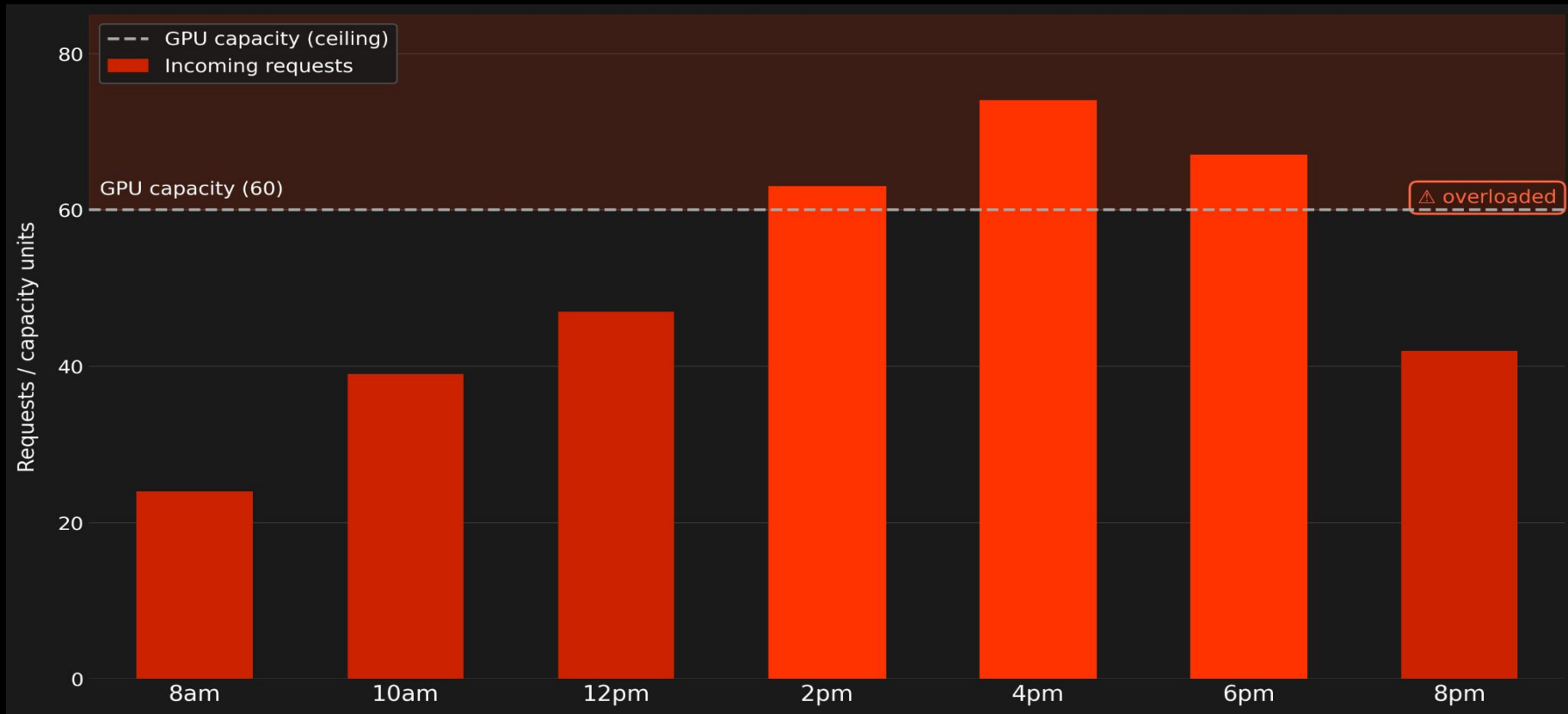
Strategy A (local cache)

Strategy B (Remote fetch)

SLO Gate and Hot-spot Migration

# Overload-oriented Scheduling

The Problem: At peak, incoming requests exceeds GPU capacity



# Overload-oriented Scheduling: Problem

## Why disaggregation matters

### vLLM — Coupled

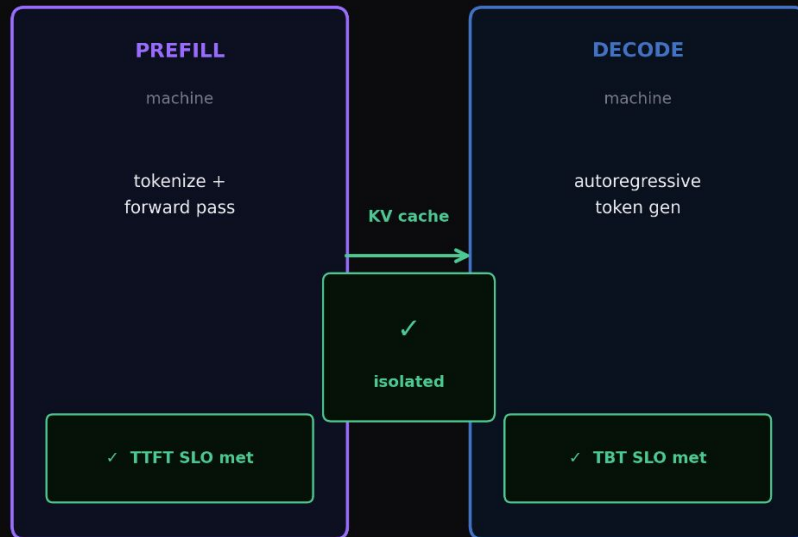
prefill & decode share the same GPU



load = GPU utilization

### Mooncake — Disaggregated

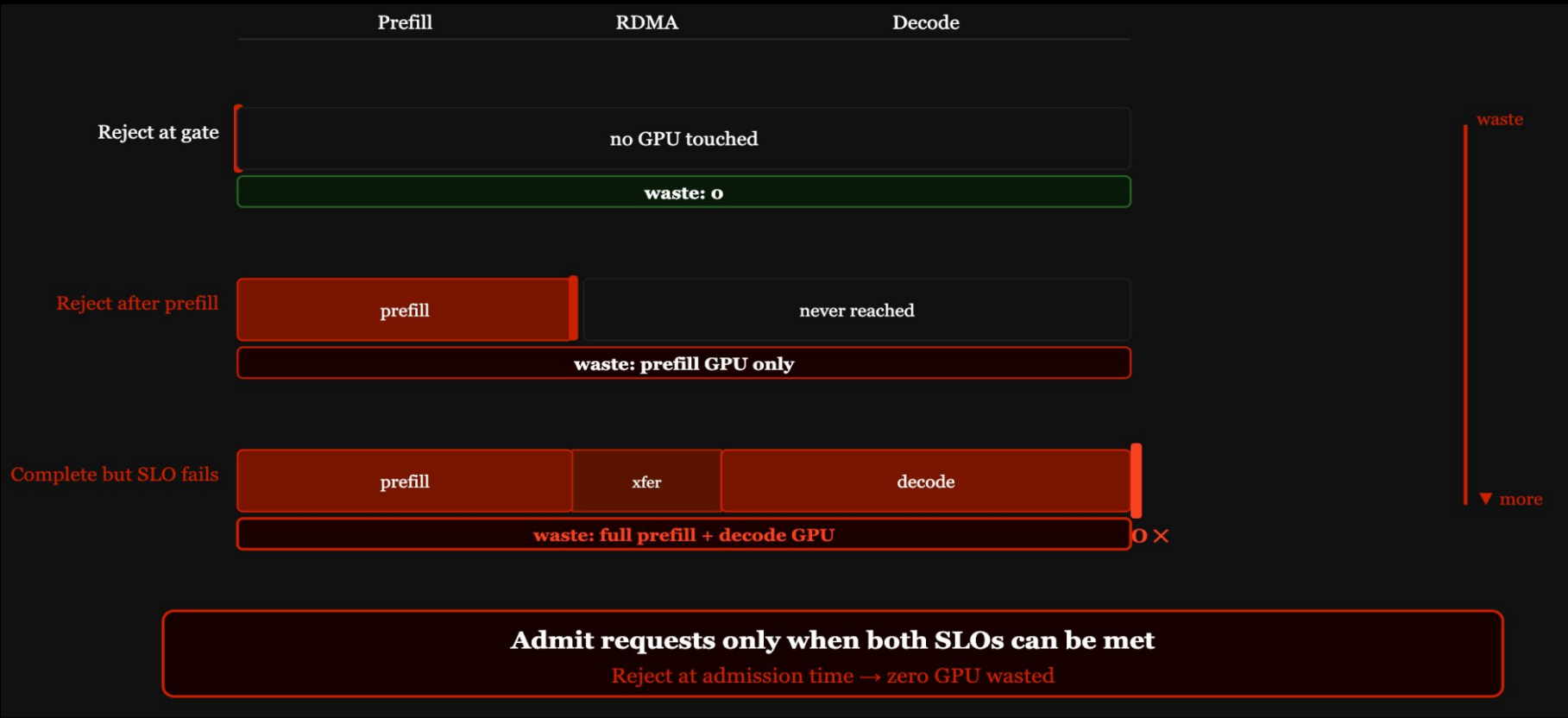
prefill & decode on separate machines



vs

load = are SLOs satisfied?

# Overload-oriented Scheduling: Problem



# Overload-oriented Scheduling: Solution

ADMISSION CONTROL · **REJECT EARLY = ZERO WASTE**

**The later you reject, the more GPU you waste**

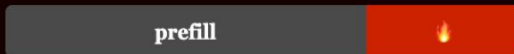
**REJECT AT GATE**



**0 waste**

No GPU touched

**REJECT AFTER PREFILL**



**partial**

Prefill GPU burned

**REJECT AT DECODE**



**all**

Everything wasted

**Mooncake: always reject at the gate** — before any GPU work begins

# Overload-oriented Scheduling: Solution

✓ check decode  
before prefill starts

**Naive fix: check decode capacity before starting prefill**

If decode is overloaded → reject early. Looks smart.

$t = 0$

prefill running (5 – 30 s)

$t + \text{prefill\_dur}$   
request hits decode

**Conductor**  
checks decode load

**Decode pool at  $t = 0$**

■ busy   ■ free

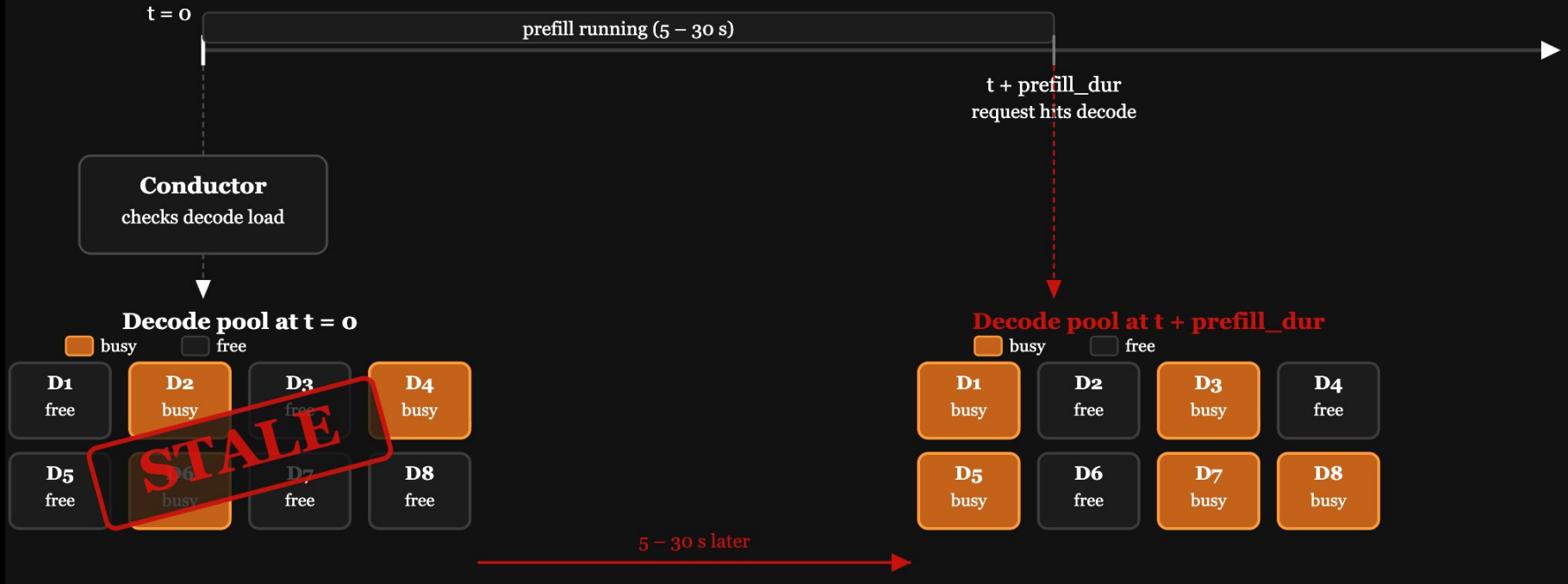


# Overload-oriented Scheduling: Solution

✓ check  
before pr

**The bug: the decode load you checked is already stale**

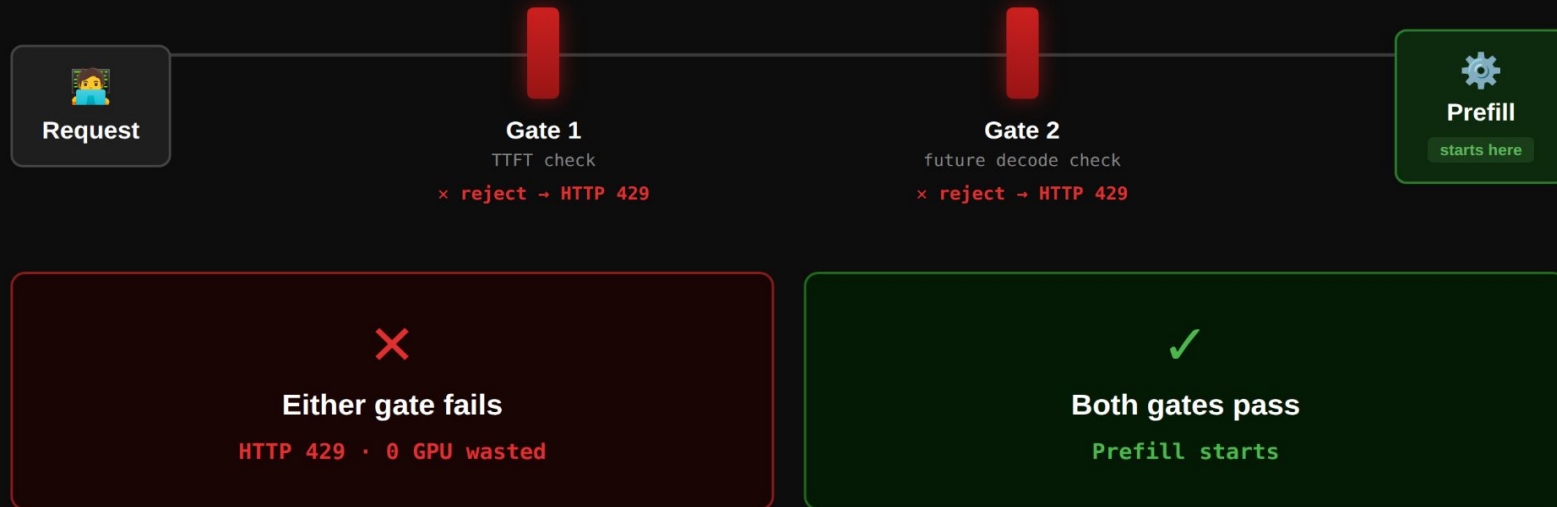
By the time the request arrives, the pool has completely changed



# Overload-oriented Scheduling: Conductor

CONDUCTOR · TWO ADMISSION GATES

Every request passes two checks before any GPU is touched



# 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**

finishes at  $t = 3$

**Job B**

finishes at  $t = 5$

**Job C**

finishes at  $t = 8$

### CURRENTLY DECODING JOBS

**Job X**

started  $t = -8$

**Job Y**

started  $t = -3$

**Job Z**

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 A**

finishes  $t = 3$

✓  $3 \leq 5 \rightarrow$  **ADD**

**Job B**

finishes  $t = 5$

✓  $5 = 5 \rightarrow$  **ADD**

~~**Job C**~~

~~finishes  $t = 8$~~

~~✗  $8 > 5 \rightarrow$  **SKIP**~~

### STEP 2 — Which decode jobs finish before $t = 5$ ? $\text{age} = 5 - \text{start\_time}$ . $\text{age} > 10s \rightarrow$ remove

~~**Job X**~~

~~started  $t = -8$~~

~~$\text{age} = 5 - (-8) = 13s > 10s$~~

~~✗ **REMOVE**~~

**Job Y**

started  $t = -3$

$\text{age} = 5 - (-3) = 8s < 10s$

✓ **KEEP**

**Job Z**

started  $t = -1$

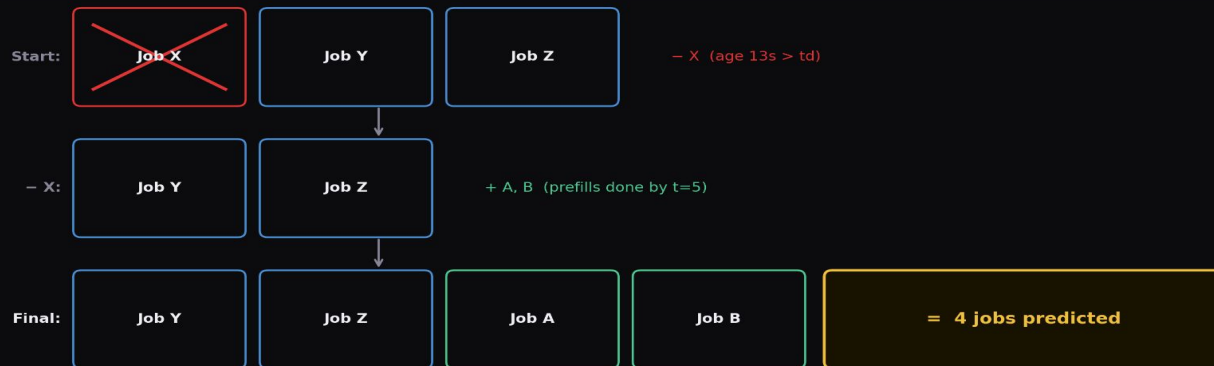
$\text{age} = 5 - (-1) = 6s < 10s$

✓ **KEEP**

## Steps 3 & 4 — Compute Pool → Reject

Build the predicted pool, then check against capacity.

### STEP 3 — Build predicted decode pool at $t = 5$



### 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.***

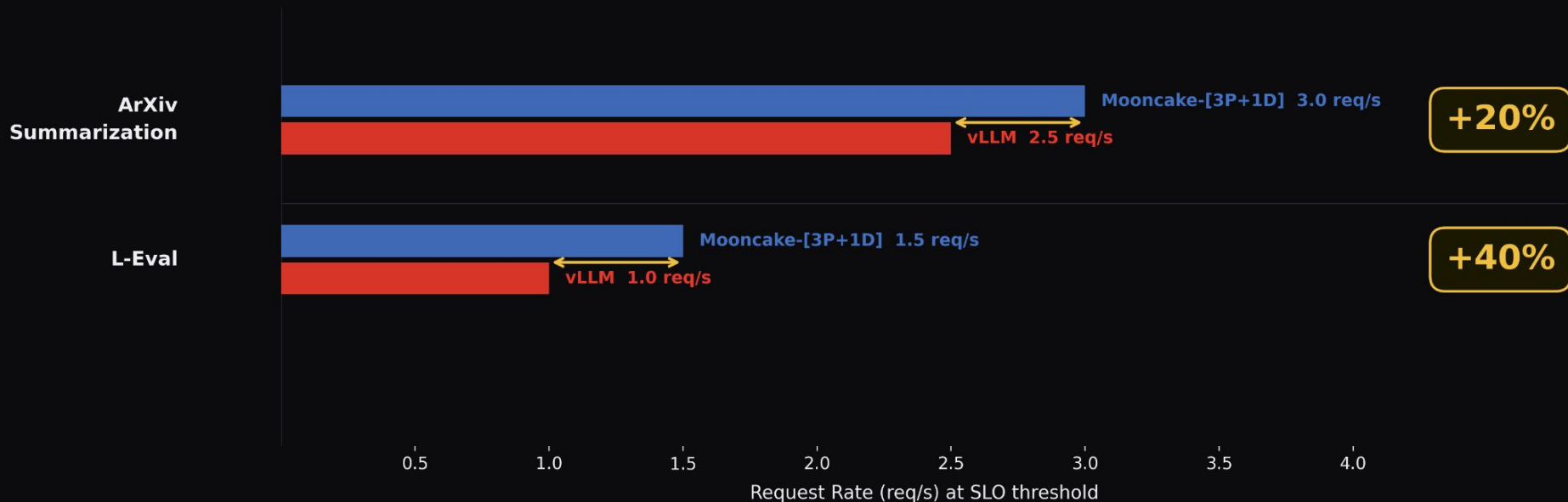
# Evaluation: Set Up Overview

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

## ① Throughput before SLO breach

Max request rate (req/s) each system sustains while meeting latency SLO



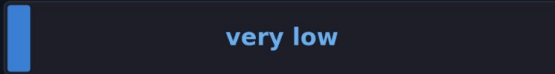
# Experiment 1: Results

## ② Why L-Eval gains more: prefix caching

Higher cache reuse → less prefill work → higher sustainable throughput

### ArXiv Summarization

cache reuse

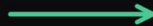


Every request unique.  
No shared prefixes.

**+20%**

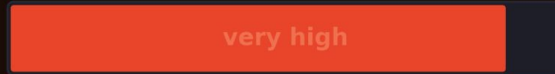
throughput gain  
over vLLM

more  
caching



### L-Eval

cache reuse



Long-context benchmark.  
Heavy prefix reuse.

**+40%**

throughput gain  
over vLLM

# Experiment 1: Results

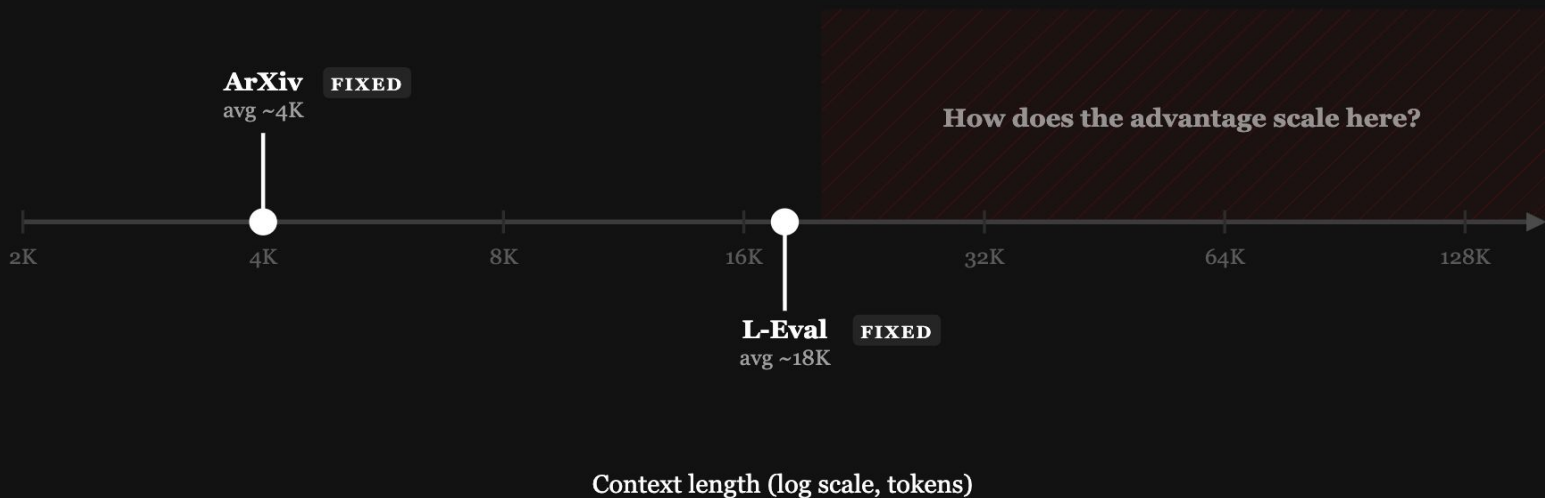
## ③ The 2P+2D tradeoff

More decoding capacity lowers TBT — but starves prefill, raising TTFT

	TTFT	TBT
Mooncake [3P+1D]	 meets SLO	 acceptable
Mooncake [2P+2D]	 breaches SLO earlier	 stays flat under load

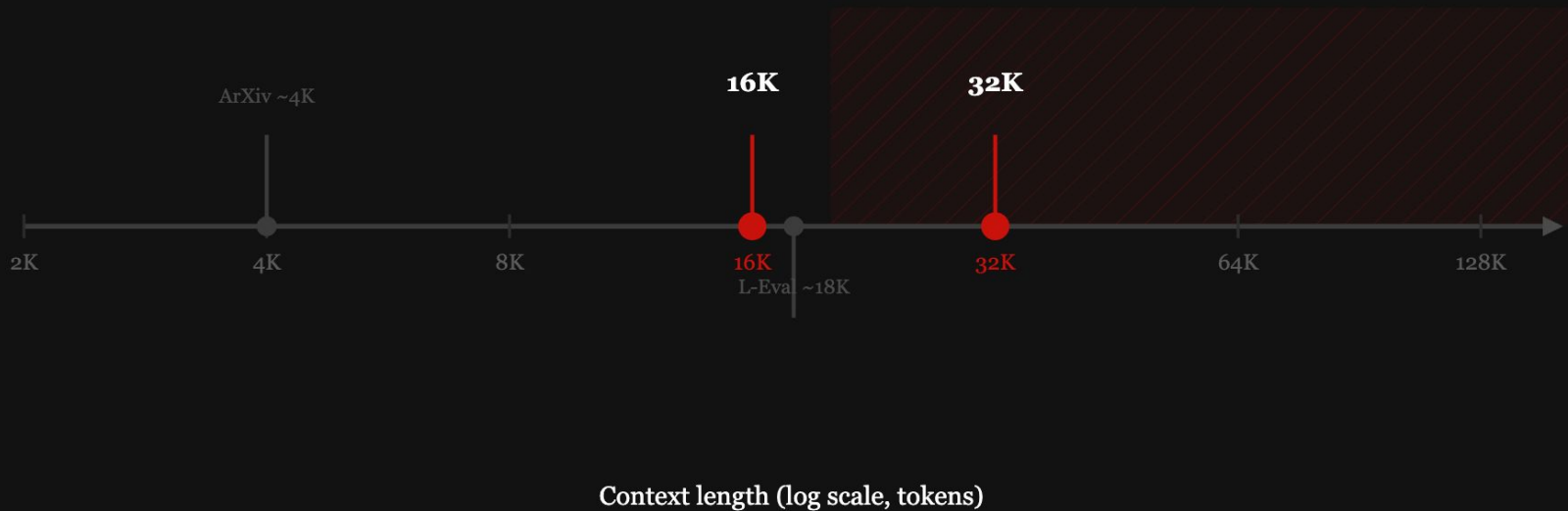
# Experiment 2: Simulated Data

DATASET SELECTION · **THE UNEXPLORED RANGE**



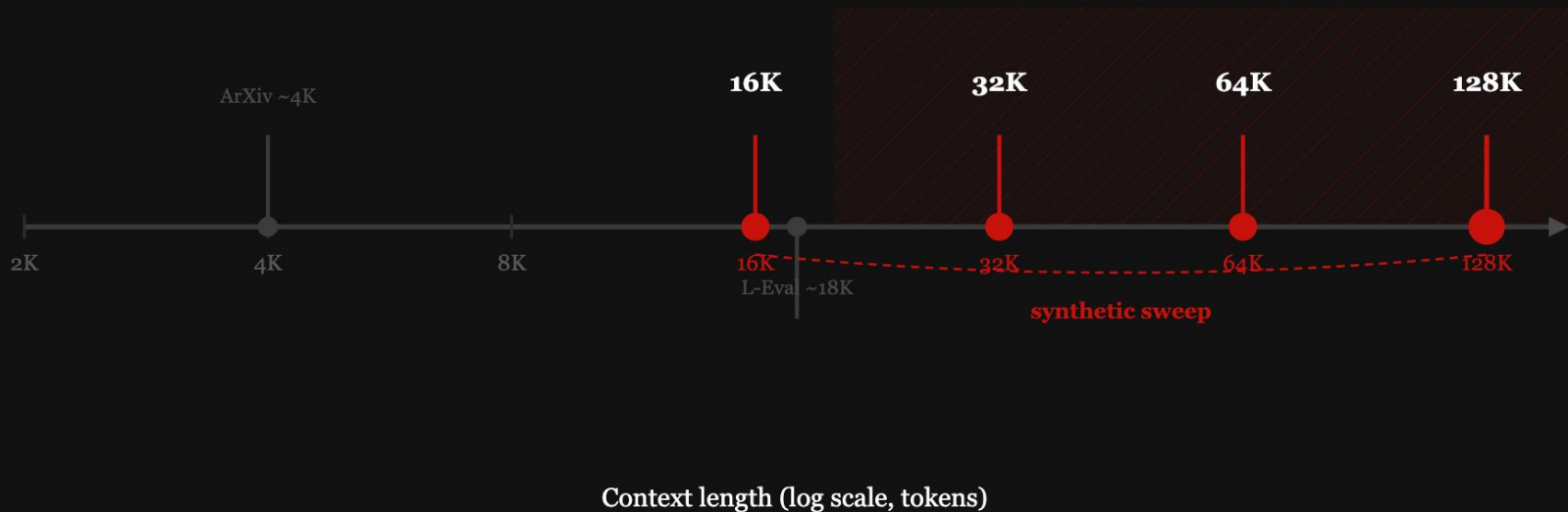
# Experiment 2: Simulated Data

DATASET SELECTION · **SYNTHETIC SWEEP**



# Experiment 2: Simulated Data

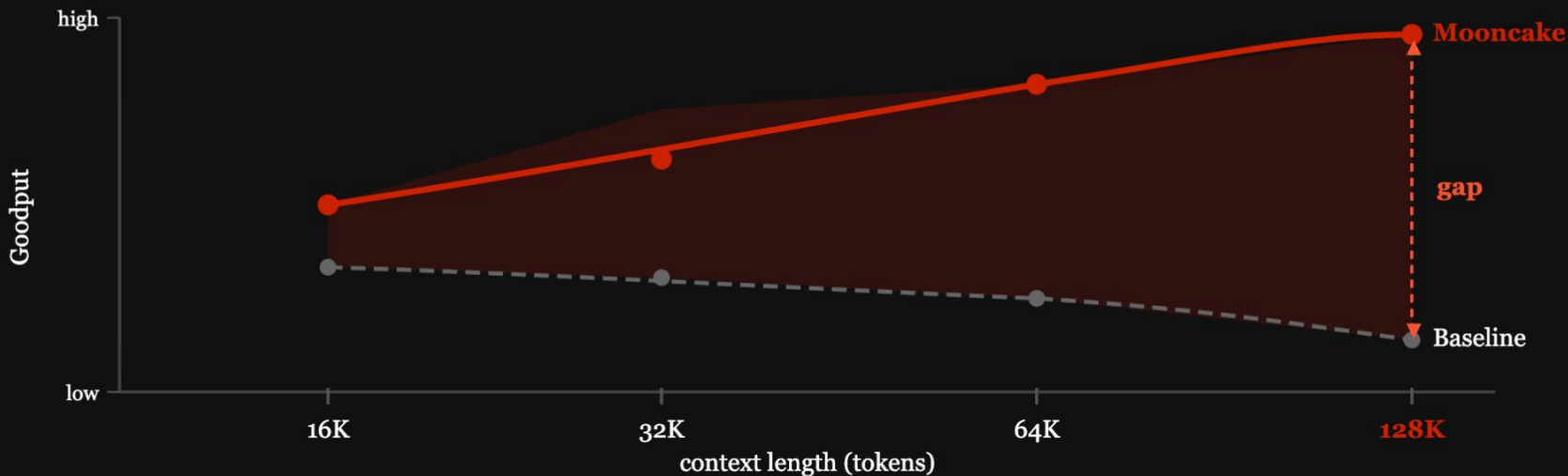
DATASET SELECTION · **SYNTHETIC SWEEP — COMPLETE**



# Experiment 2: Results

EXPECTED RESULT · **GAP GROWS WITH CONTEXT LENGTH**

**Longer context → larger goodput advantage for Mooncake**



**Disaggregation + prefix caching advantage widens as context grows**

128K = extreme end — largest prefill cost, highest KVCache value, biggest expected gain

# Experiment 3: Real Workload

## Mooncake



10 Prefill nodes

10 Decode nodes

Disaggregated  
prefill-decode

VS

## vLLM

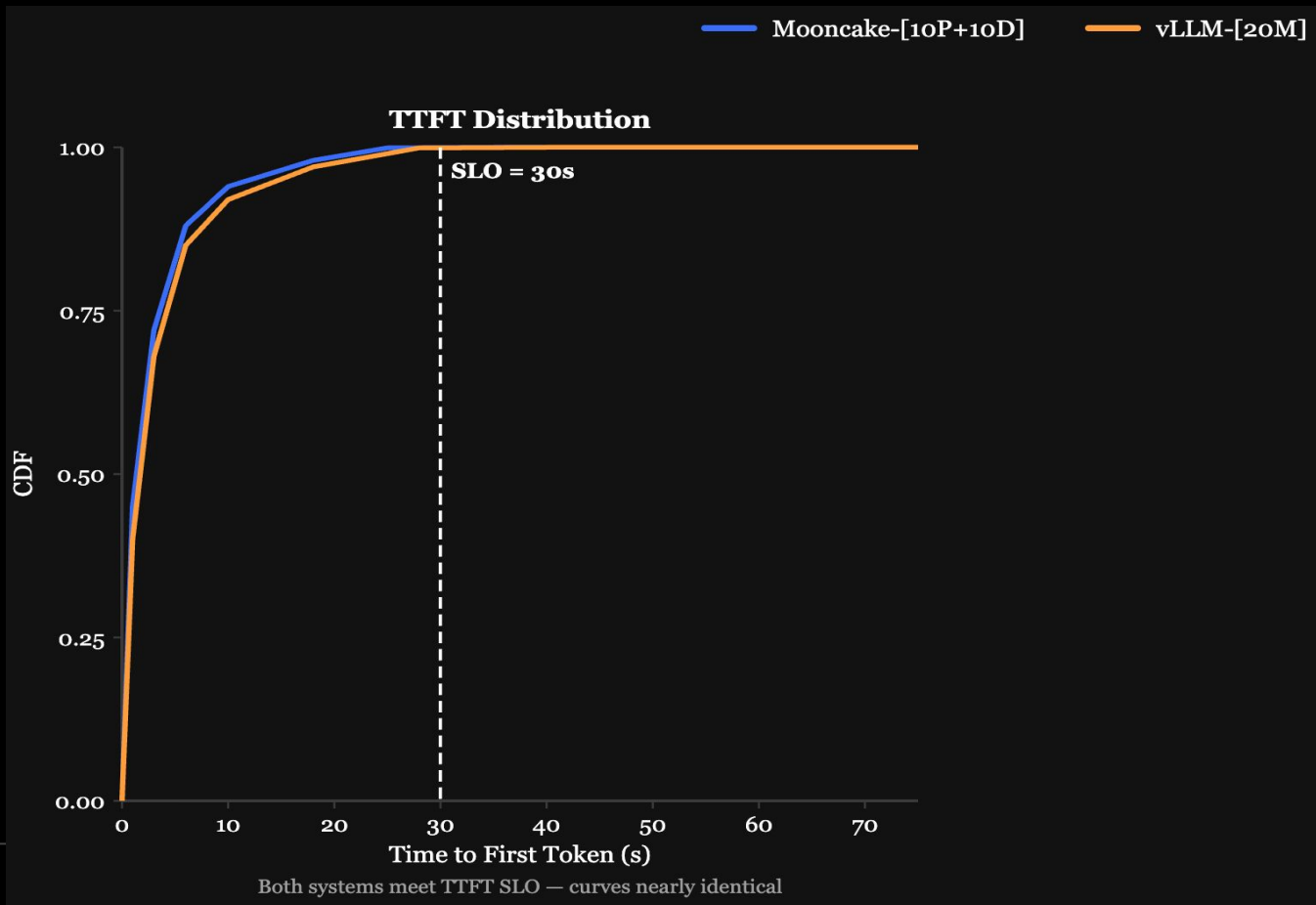


20 Monolithic instances

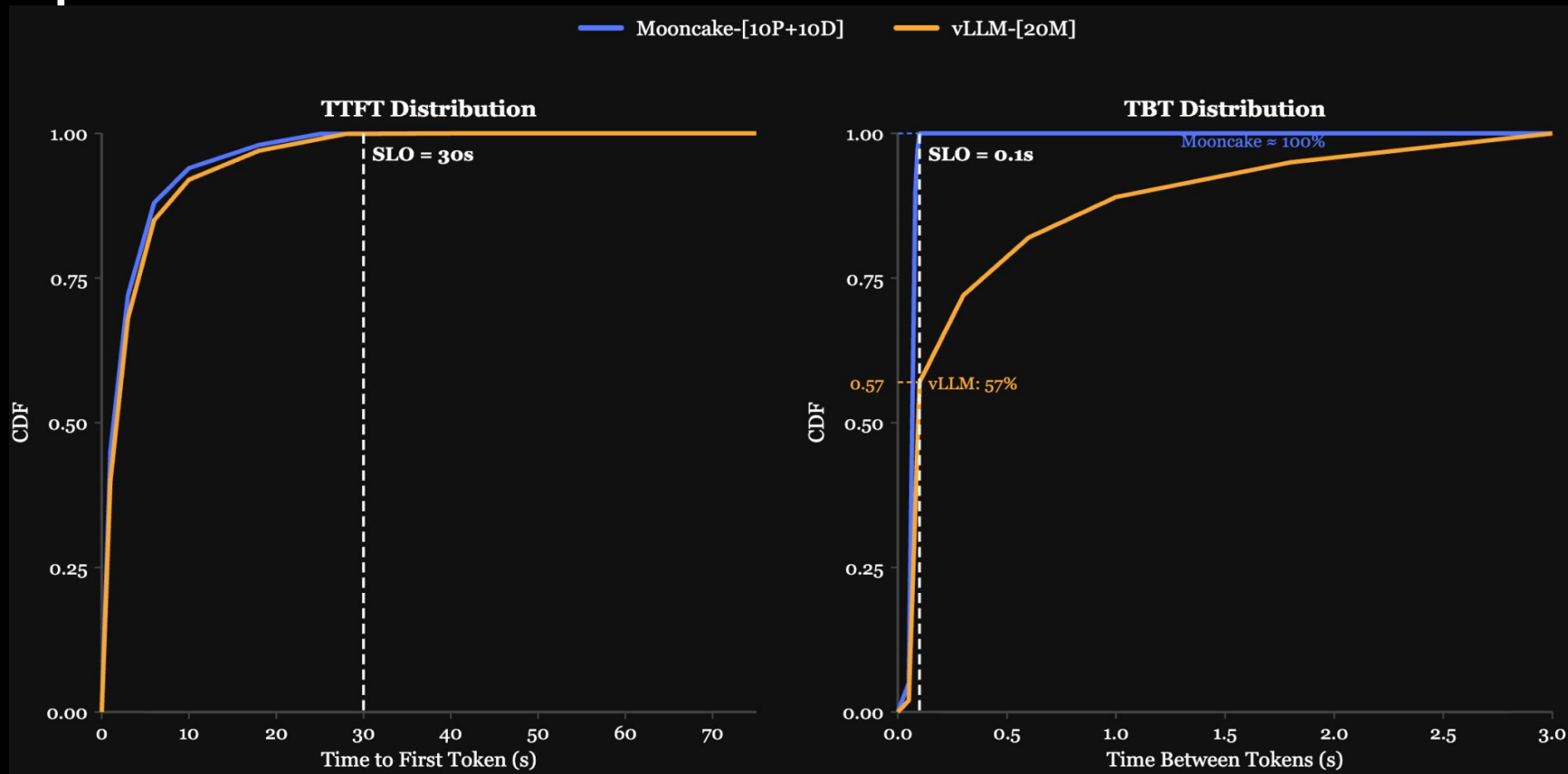
Collocated  
prefill + decode

Same total GPU resources · Real request traces replayed

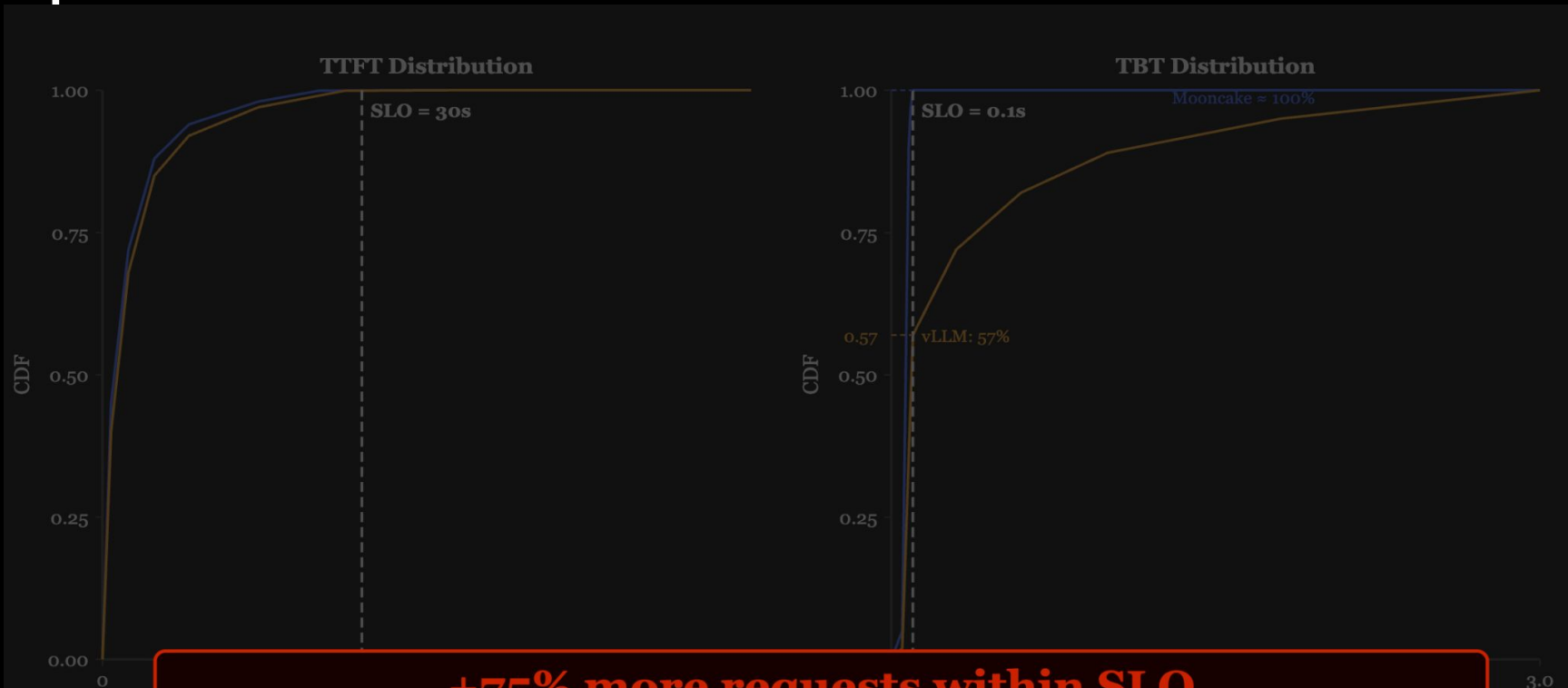
# Experiment 3: Results



# Experiment 3: Results



# Experiment 3: Results



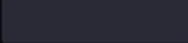
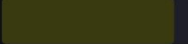
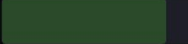
**+75% more requests within SLO**

Mooncake ~100% TBT compliance · vLLM only 57% · TTFT identical for both

# 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

## Throughput optimization

FasterTransformer

TensorRT-LLM

Deepspeed  
Inference

Splitwise

## Scheduling

Orca

SARATHI

FastServe

FlexGen

DistServe

TetralInfer

## KV-Cache Management

vLLM

Prompt  
Cache

SGLang

AttentionSore

Mooncake

KV-cache centric disaggregated  
arch, building on open-source vLLM,  
Radix-attention closest to Mooncake



# Future work

