State of vLLM 2026

Ray Summit 2026 Recap

Engine, ecosystem and project direction

Talk by Woosuk Kwon and Zachary Xi
Recap by Paphanin Amnuaikitwanit (Toon)

Open models and inference ownership

≈10 days

between open-model releases

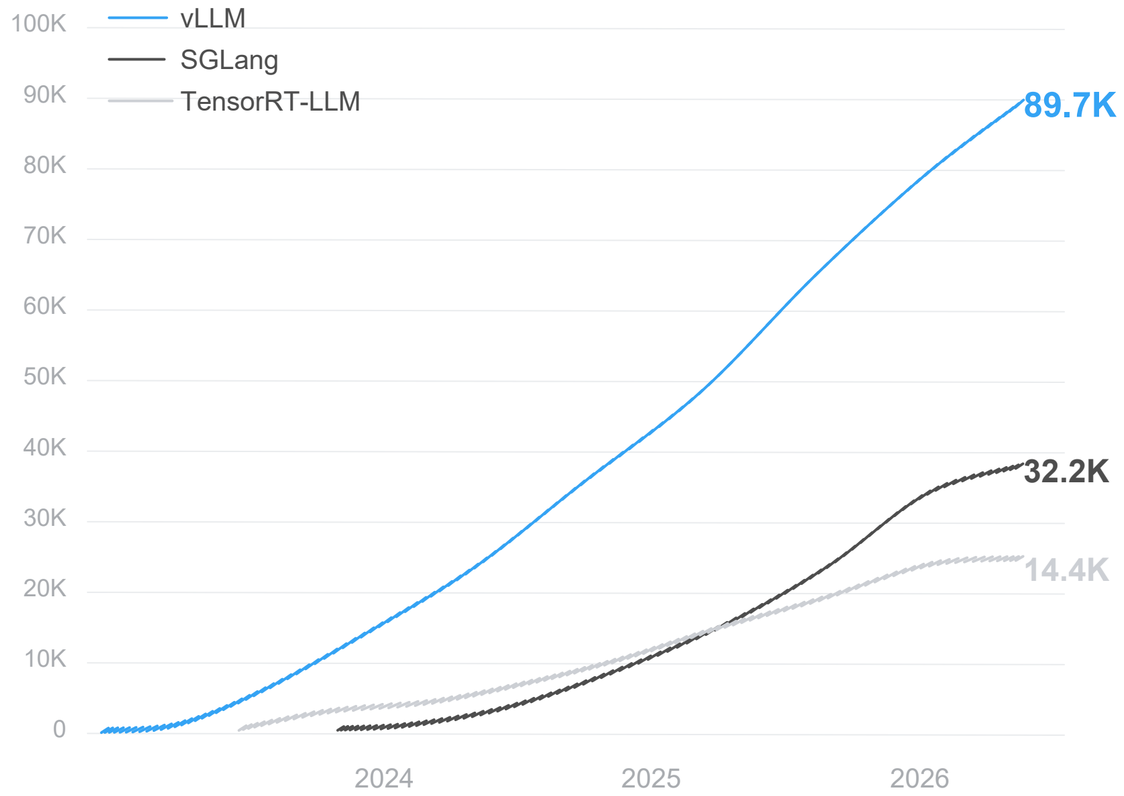
Inference capacity

Cost and privacy

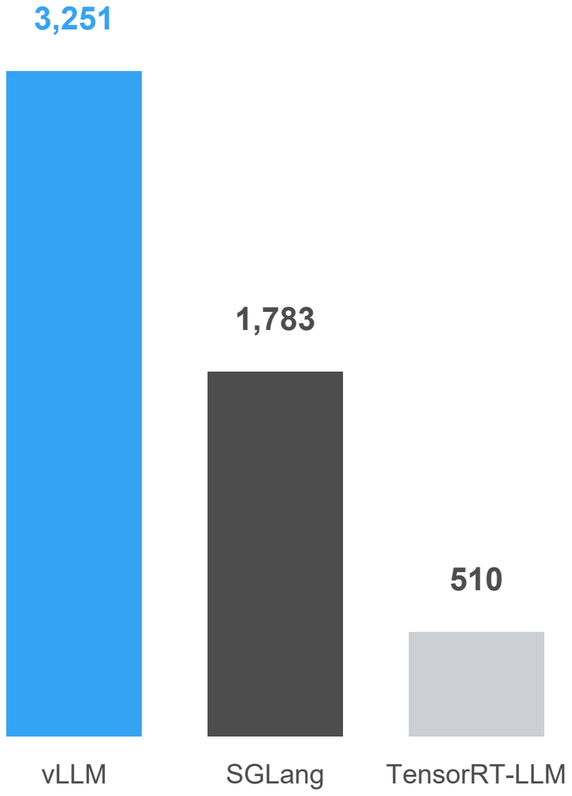
Custom weights and workloads

A growing vLLM community

GitHub stars over time



Contributors



89.7K

GitHub stars

3,251

contributors and counting

Source: GitHub, August 2026

Three areas of focus

01

Core redesign

02

Model optimization

03

Agentic workloads

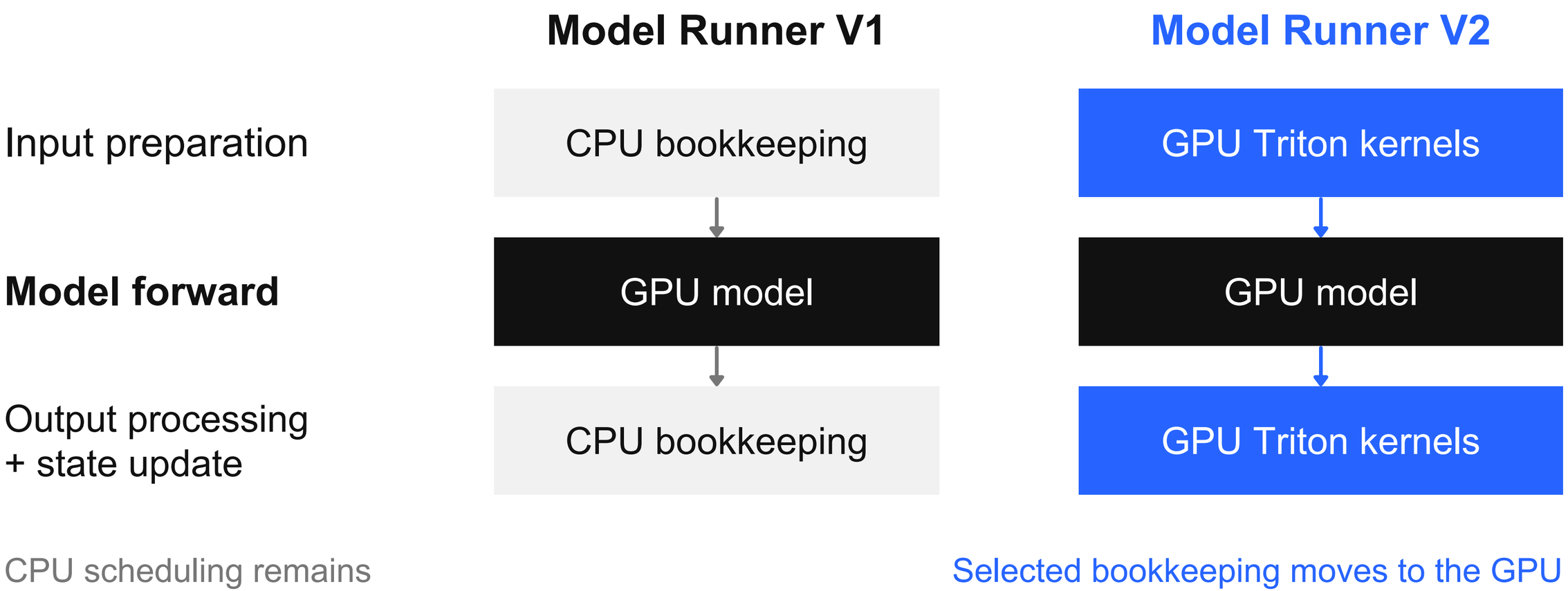
Three areas of focus

01

Core redesign

- Model Runner V2
- Asynchronous Scheduling
- ModelState Abstraction
- Rust API Service
- KV Cache Layout Standardization

1.1) Model Runner V2



CPU scheduling remains

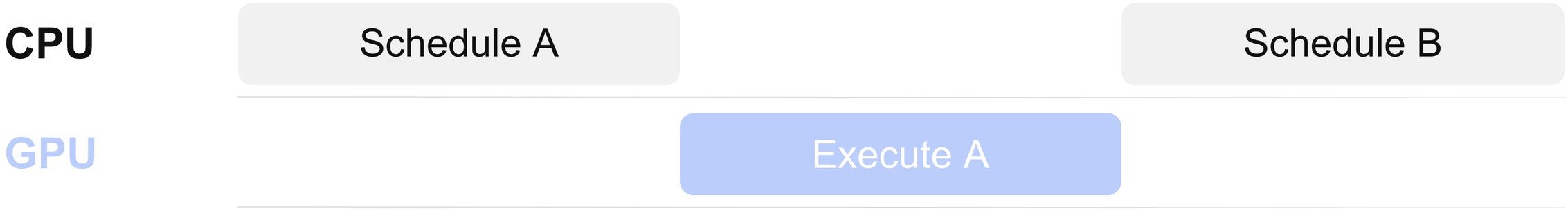
Selected bookkeeping moves to the GPU

1.1) Model Runner V2

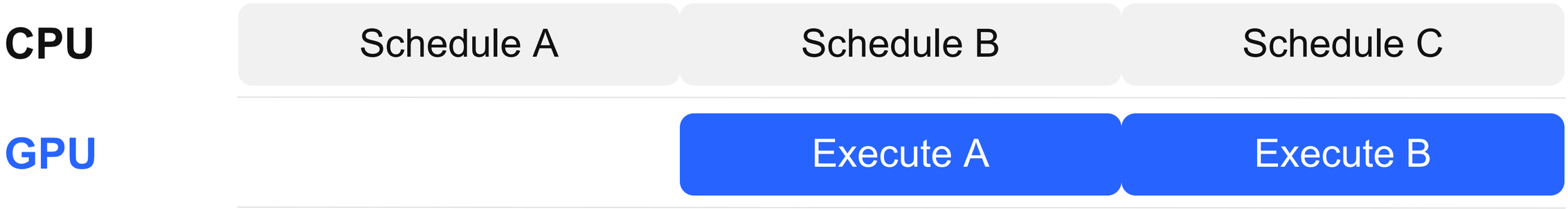


1.2) Asynchronous scheduling

Sequential execution

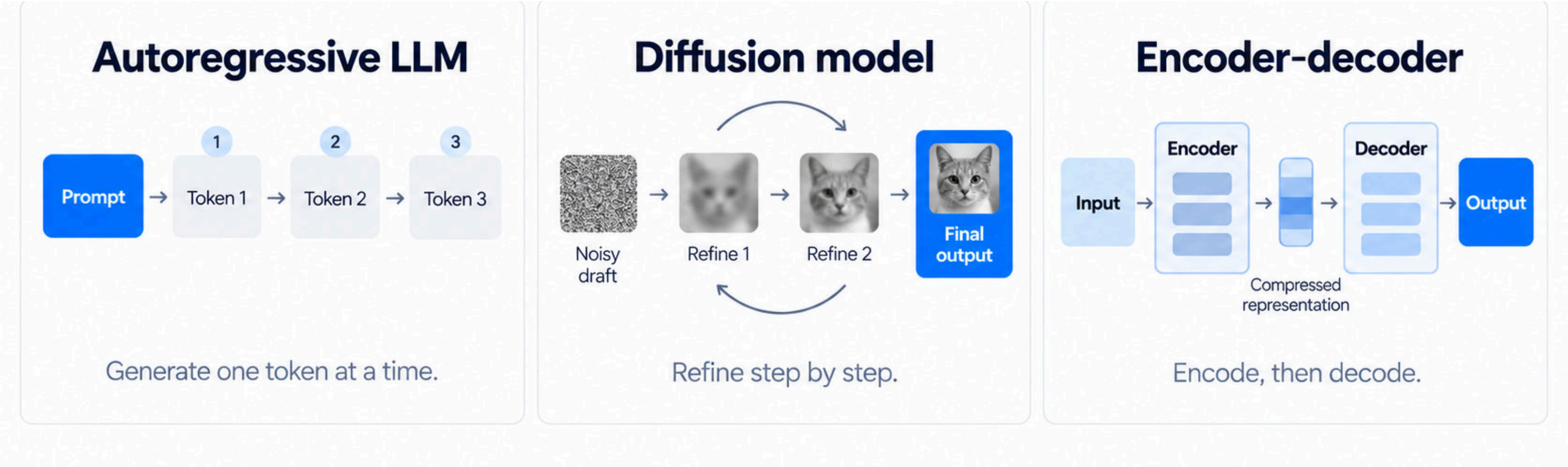


Asynchronous execution



1.3) ModelState abstraction

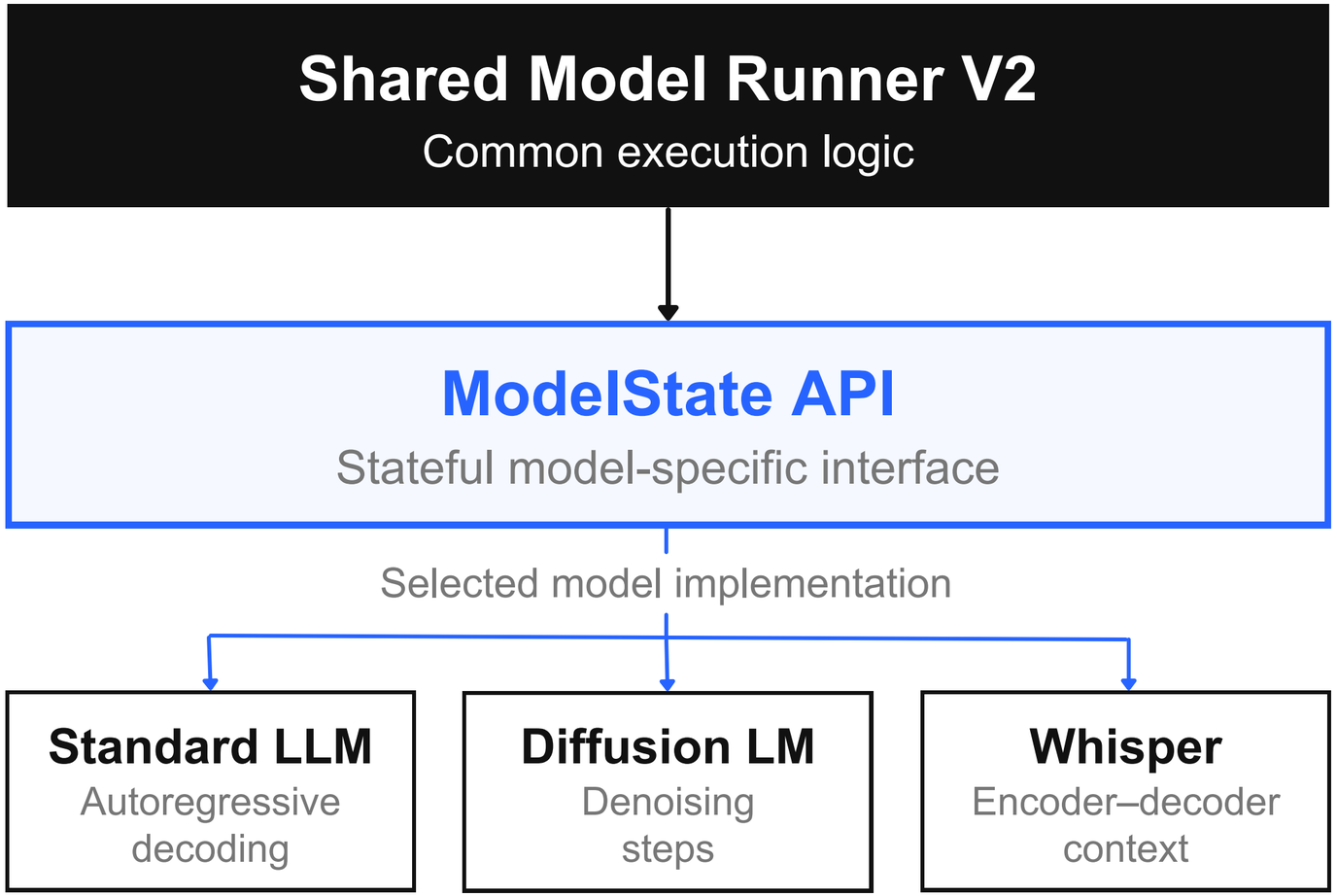
vLLM supports several architectures.



1.3) ModelState abstraction

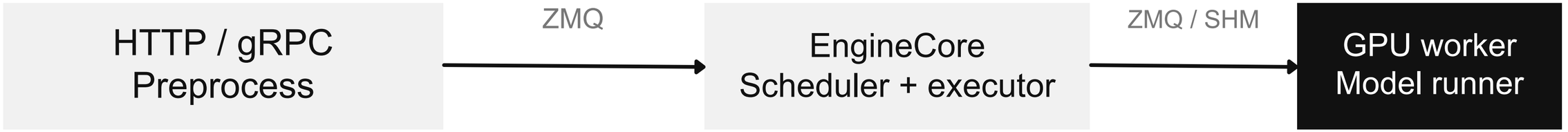
The solution

ModelState encapsulates model-specific state and behavior behind one API.

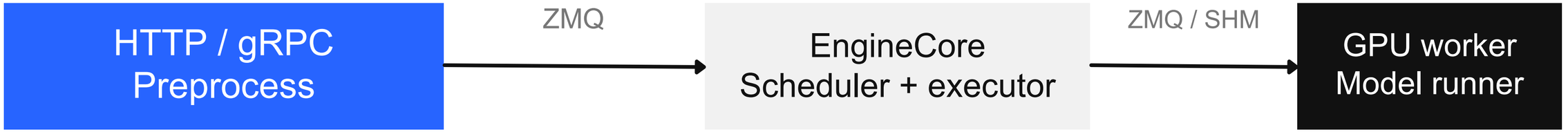


1.4) Rust API Server

Python API Server



Rust API Server



Stable P90 latency

Simple scale-out

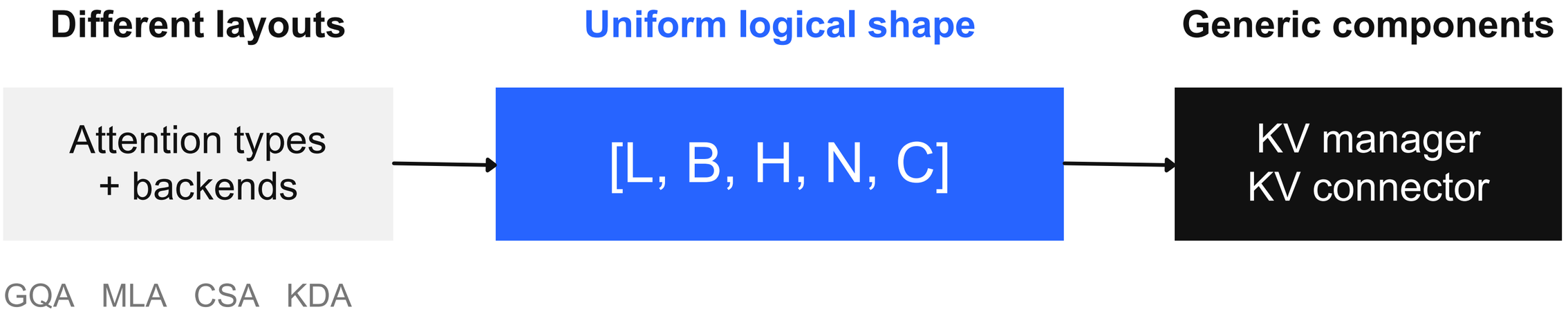
Shared Rust crates

1.4) Rust API Server

Metric	Python	Rust
Startup time	Multi-second	~7 ms
100k random prompts (input_len=8192)	Minutes	Seconds (rayon parallelism)
Peak memory at 1400 concurrency	High (GIL + per-object overhead)	<100 MB (Arc<str> prompt sharing)

github.com/vllm-project/vllm-bench

1.5) KV cache layout standardization



Three areas of focus

02

Model optimization

- GPU Kernel Fusion
- Model Parallelism
- Speculative Decoding

2.1) Why customs GPU Kernels?

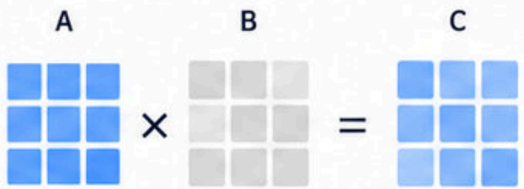
THE PROBLEM

General-purpose libraries cannot always give best latency for every model and hardware combination.

WHY GENERIC KERNELS FALL SHORT

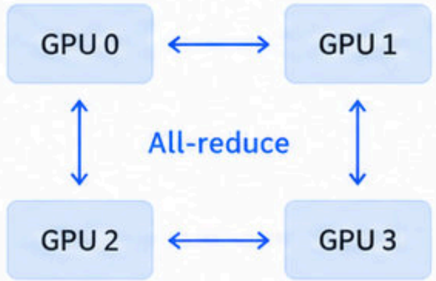
1. GEMM

Matrix multiplication on each GPU.



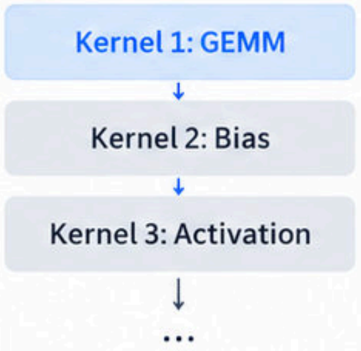
2. All-reduce

Share partial results across GPUs.



3. Separate launches

Run multiple small kernels (e.g. GEMM, bias, activation) one by one.



4. Extra overhead

Time spent on communication and kernel launches adds up, especially for long sequences and new architectures.



2.1) Kernel fusion: why it helps

Multi-GPU inference involves both computation and communication.

BEFORE

Separate operations

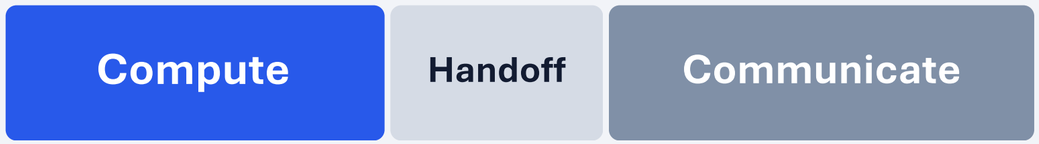
GEMM

Matrix multiply

Reduce-scatter

Share GPU results

Execution timeline



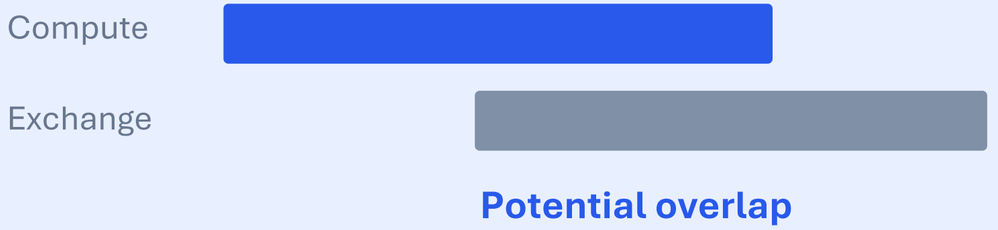
AFTER

Fused implementation

GEMM + Reduce-scatter

Coordinate compute and communication

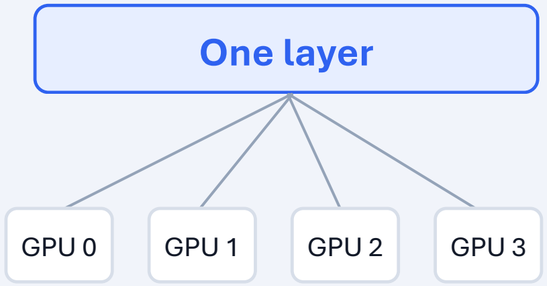
Execution timeline



2.2) Three ways GPUs can share the work

Parallelism is about what gets divided

Tensor parallelism



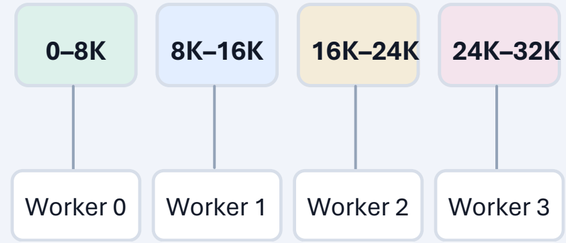
Divide computations inside a layer

Pipeline parallelism



Divide model layers into stages

Context parallelism



Divide long input tokens across workers

Different models benefit from different ways of splitting the work.

Why TP worked well before

Especially for many earlier dense transformer models

Why it fit older models

- Dense matrix computation was the main workload
- Work split cleanly across GPUs
- Communication cost was acceptable relative to compute
- So TP was often a natural default choice

Example: Llama-style models

TP helped because the compute benefit was bigger than the communication cost.

Why TP is not always enough now

Modern models often need a mix of parallelism strategies

Why TP alone may struggle

- Some work does not split cleanly by heads or tensors
- Communication cost can become more noticeable
- Long-context and sparse-attention workloads may need other splits
- MoE and newer architectures introduce different bottlenecks

Composable parallelism

Tensor (TP)
Pipeline (PP)
Data (DP)
Expert (EP)

X

Sequence (SP)
Prefill context (PCP)
Decode context (DCP)

2.2) GLM-5.2: TP vs PCP

GLM 5.2 = sparse-attention

Tensor Parallelism (TP)

INPUT PROMPT: 32K TOKENS

GPU 0	GPU 1	GPU 2	GPU 3
Weights: shard	Weights: shard	Weights: shard	Weights: shard
Input: 32K	Input: 32K	Input: 32K	Input: 32K
Indexer repeats	**Indexer repeats**	**Indexer repeats**	**Indexer repeats**
Head shard 0	Head shard 1	Head shard 2	Head shard 3

Prefill Context Parallelism (PCP)

0–8K	8K–16K	16K–24K	24K–32K
GPU 0	**GPU 1**	**GPU 2**	**GPU 3**
Weights: full	Weights: full	Weights: full	Weights: full
Q: chunk 0	**Q: chunk 1**	**Q: chunk 2**	**Q: chunk 3**
KV shard 0	KV shard 1	KV shard 2	KV shard 3

KV all-gather for attention (MLA backend)

~3.5× faster TTFT vs TP in the reported test

2.3) Why speculative decoding?

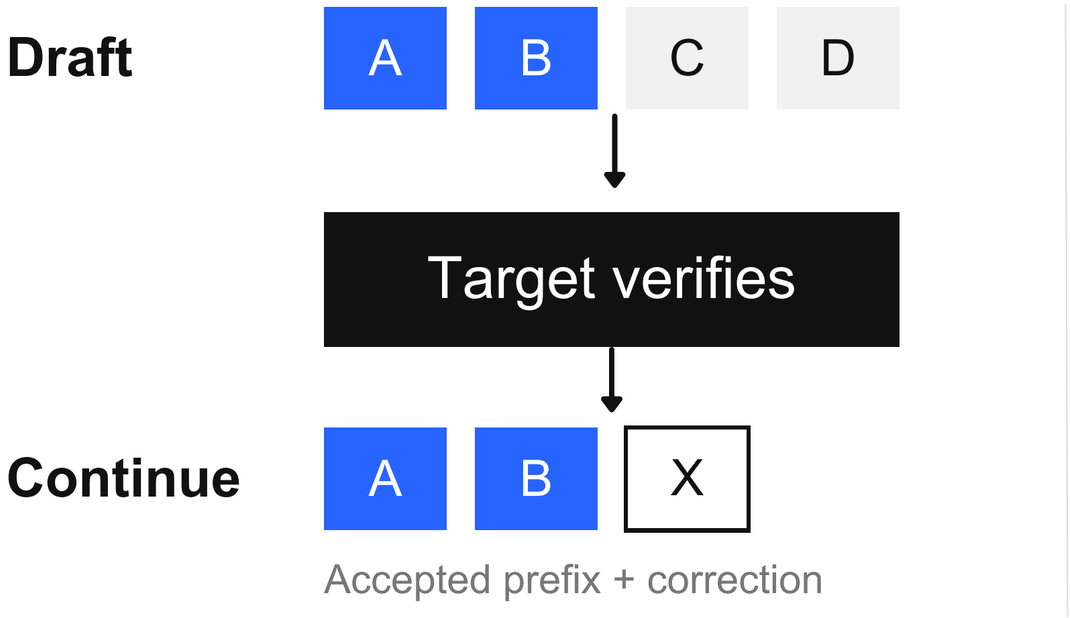
THE PROBLEM

Autoregressive decoding usually needs a model step for every next token.

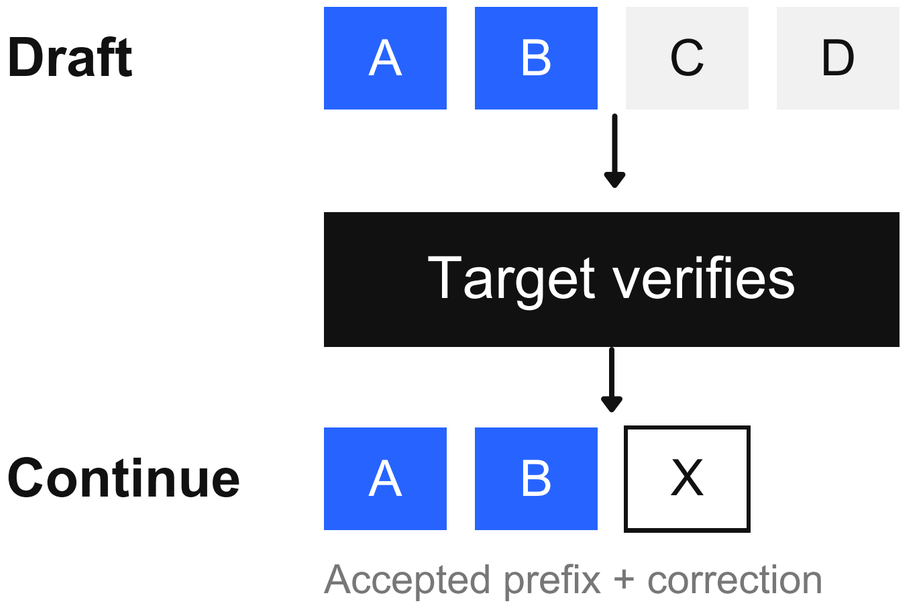
WHY DECODING IS SLOW

1 token → 1 model step → Repeat → Higher latency

2.3) Speculative decoding



2.3) Speculative decoding



Supported methods

Eagle

Single- / multi-head MTP

DFlash 1 & 2

DSpark: confidence-based scheduling

RecoverSSM: linear attention

Compatibility Logprobs Structured outputs PD disaggregation

Training integration: Speculators + TorchSpec

Three areas of focus

03

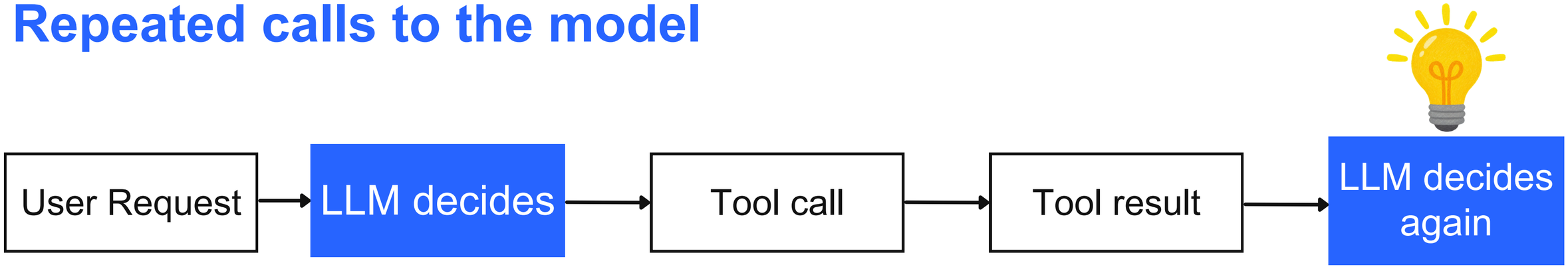
Agentic workloads

- KV Reuse
- Mooncake
- Session-Aware Routing
- Smart Cache Eviction
- Sparse Retention

3) Agentic workload focus

One user task can require several inference requests

Repeated calls to the model



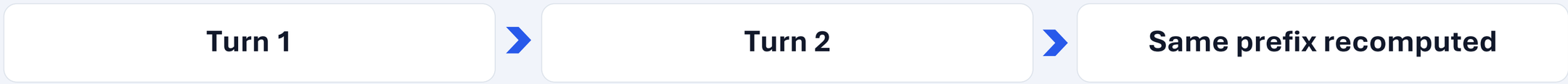
At the second LLM call, much of the earlier context is unchanged.

3.1) Why reuse KV cache?

THE PROBLEM

Each new agent turn may repeat much of the same context and pay for prefill again.

WHAT GETS REPEATED



3.1) Multi-turn KV reuse

The reusable prefix grows as the session progresses

Turn 1	**Common prompt**	**New A**
Turn 2	**Common prompt + A**	**New B**
Turn 3	**Common prompt + A + B**	**New C**

3.2) Why Mooncake?

THE PROBLEM

GPU memory is limited; useful KV states may need to survive across requests or GPU workers.

WHY LOCAL GPU CACHE IS NOT ENOUGH

Useful KV → **GPU memory limit** → **Cache may be lost**

3.2) Mooncake integration

1 Prefill sends KV

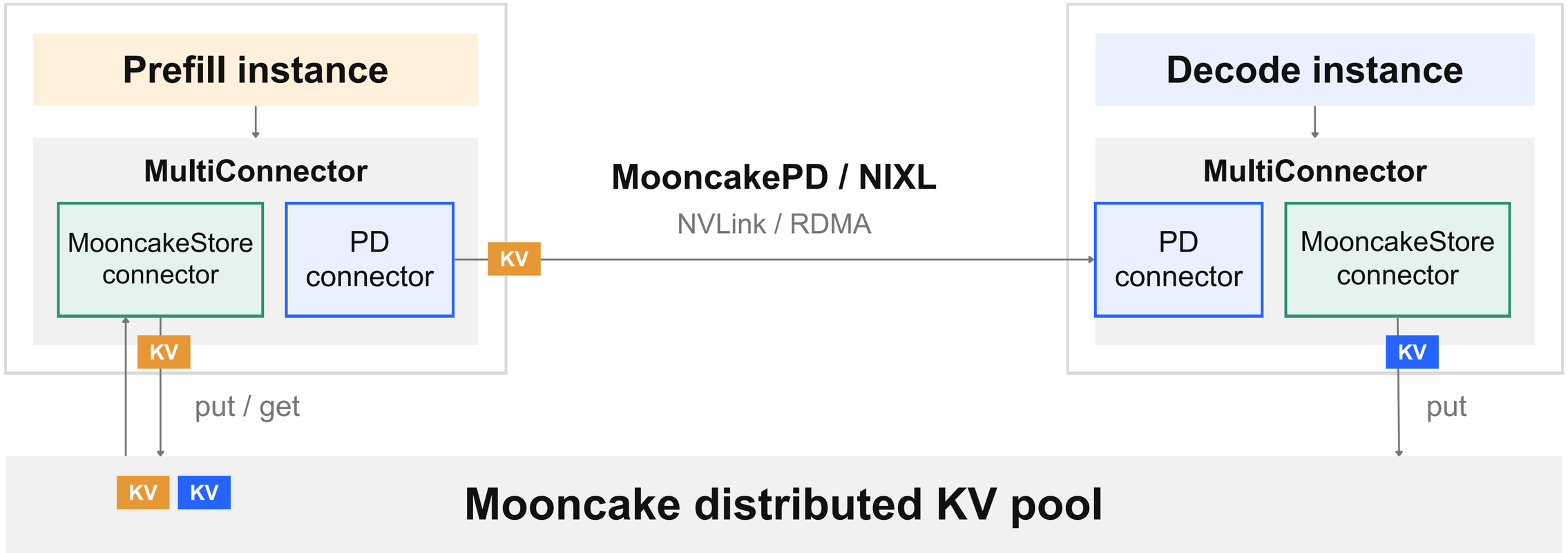
PD transfer + pool write

2 Decode stores KV

New KV joins the pool

3 Next prefill reuses

Load both KV segments



3.3) Why session-aware routing?

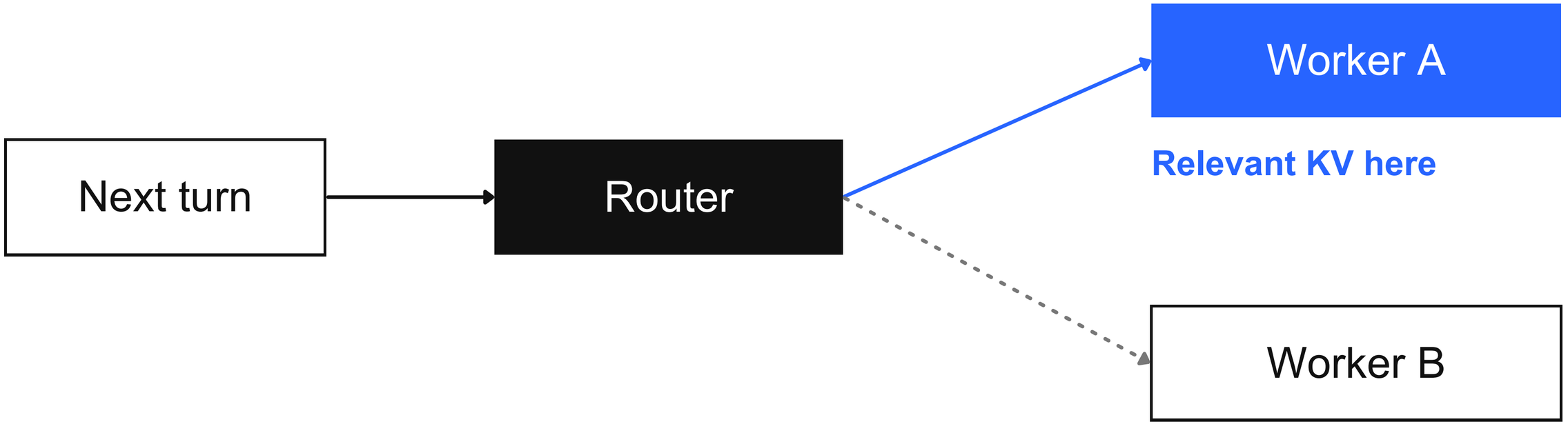
THE PROBLEM

A worker may receive the next turn even though another worker has its useful cached prefix.

WHERE LOCALITY IS LOST

Next turn → **Worker A has KV** → **Request goes elsewhere**

3.3) Session-aware routing



Dynamo **llm-d** **SMG**

3.4) Why smarter cache eviction?

THE PROBLEM

Saving every intermediate KV state wastes memory, especially when future turns omit some reasoning tokens.

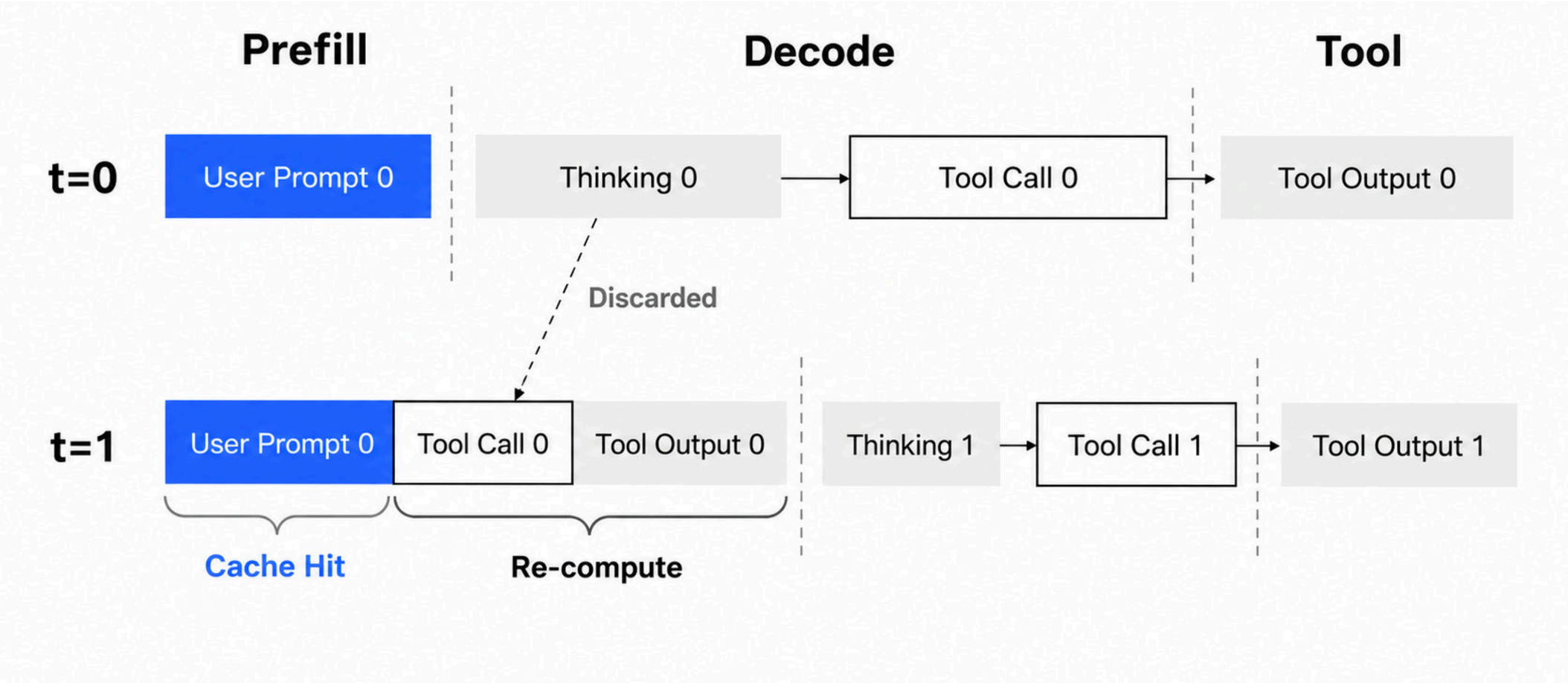
WHY NAIVE RETENTION WASTES MEMORY

Save every state → **Memory grows** → **Many states never reused**

3.4) Sparse retention

Sliding-window and linear-attention state

When the next prompt omits thinking



Three areas of focus

01

Core redesign

- Model Runner V2
- Asynchronous Scheduling
- ModelState Abstraction
- Rust API Service
- KV Cache Layout Standardization

02

Model optimization

- GPU Kernel Fusion
- Model Parallelism
- Speculative Decoding

03

Agentic workloads

- KV Reuse
- Mooncake
- Session-Aware Routing
- Smart Cache Eviction
- Sparse Retention

What’s next

01 **Simpler KV management APIs**

02 **Performance metrics and models**

Granular metrics

03 **Broader hardware support**

AMD GPU, Google TPU

04 **RL and multimodal integration**