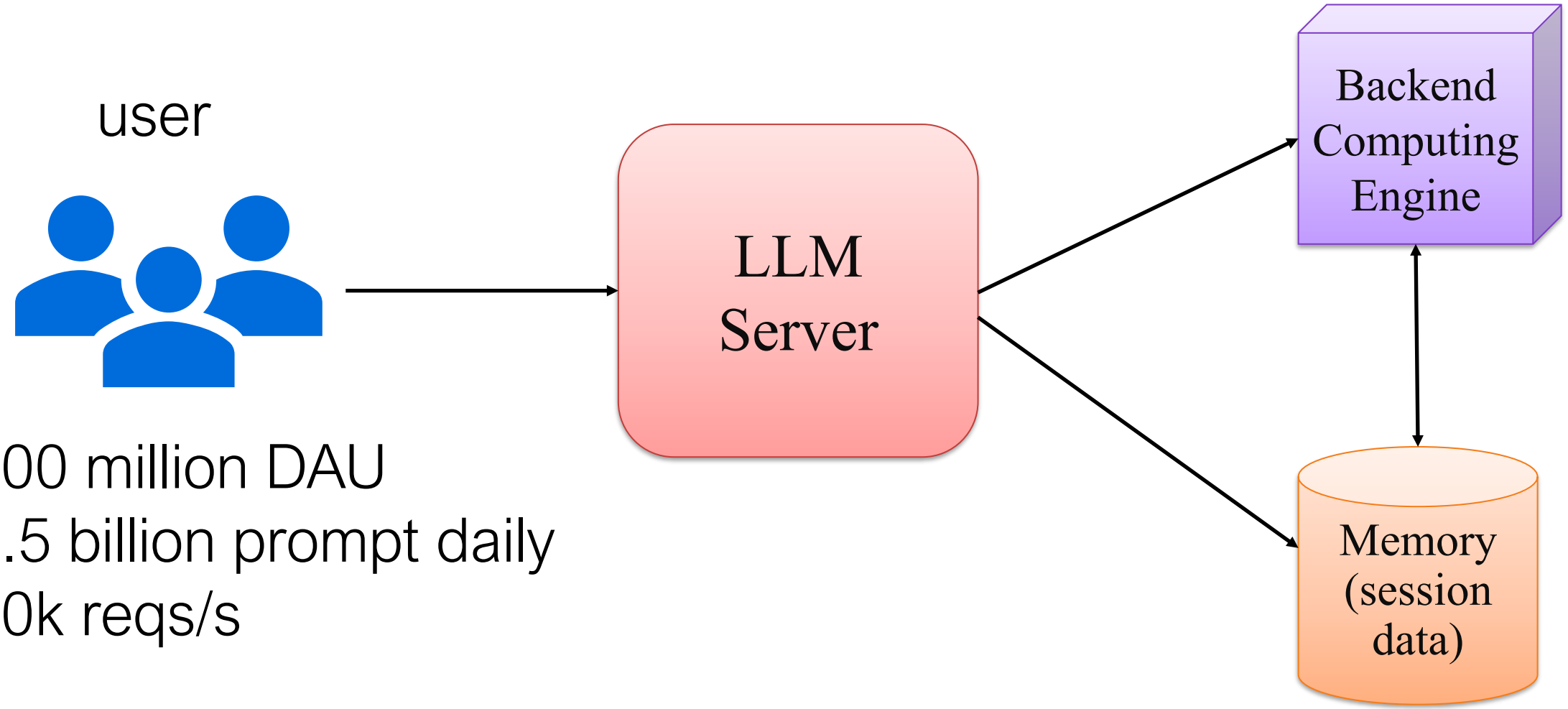


LLM Serving with RadixAttention

Lei Li

Language Technologies Institute | Carnegie Mellon University School of Computer Science

LLM Serving



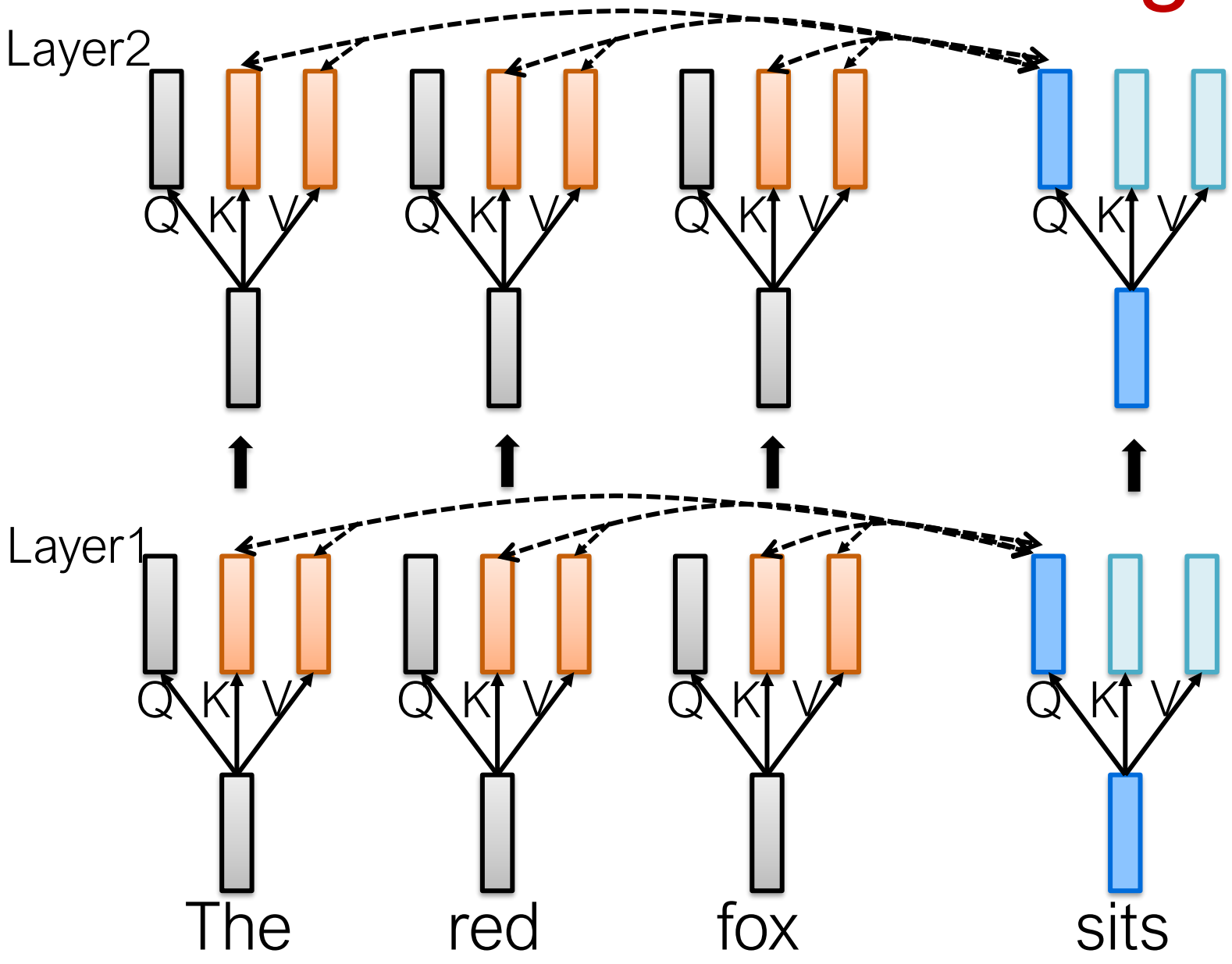
200 million DAU
2.5 billion prompt daily
30k reqs/s

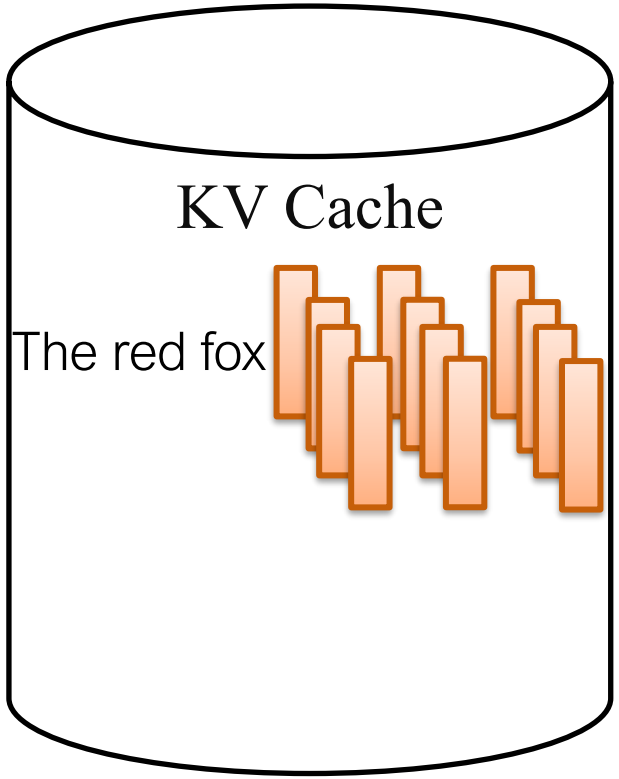
LLM Request Scheduler

1. Receive input messages
2. Stream model outputs
3. Check the stop conditions
4. Reorder requests and prepare a batch
5. Allocate memory for the next batch and running batch

```
while True:
    recv_reqs = recv_requests()
    process_input_requests(recv_reqs)
    batch = get_next_batch_to_run()
    result = run_batch(batch)
    process_batch_result(batch, result)
```

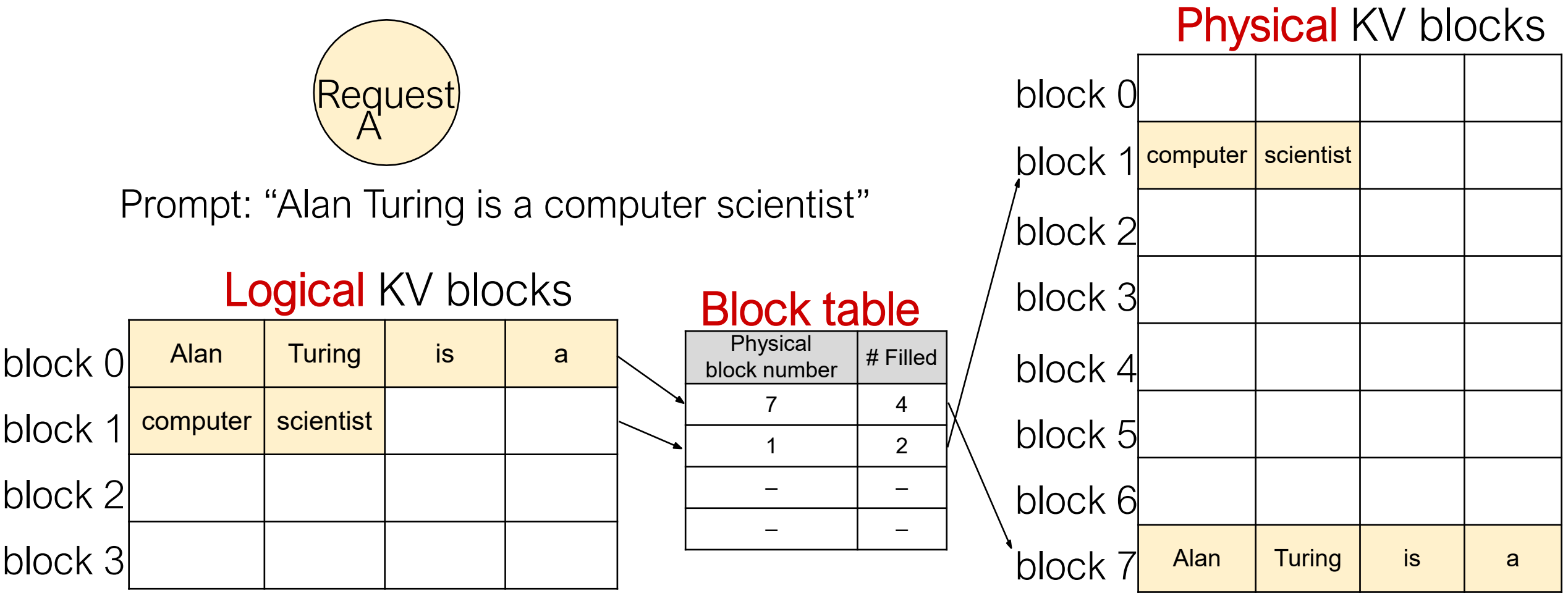
KV Cache Management





$$\text{Att} = \text{Softmax}(q \cdot K^T) \cdot V$$

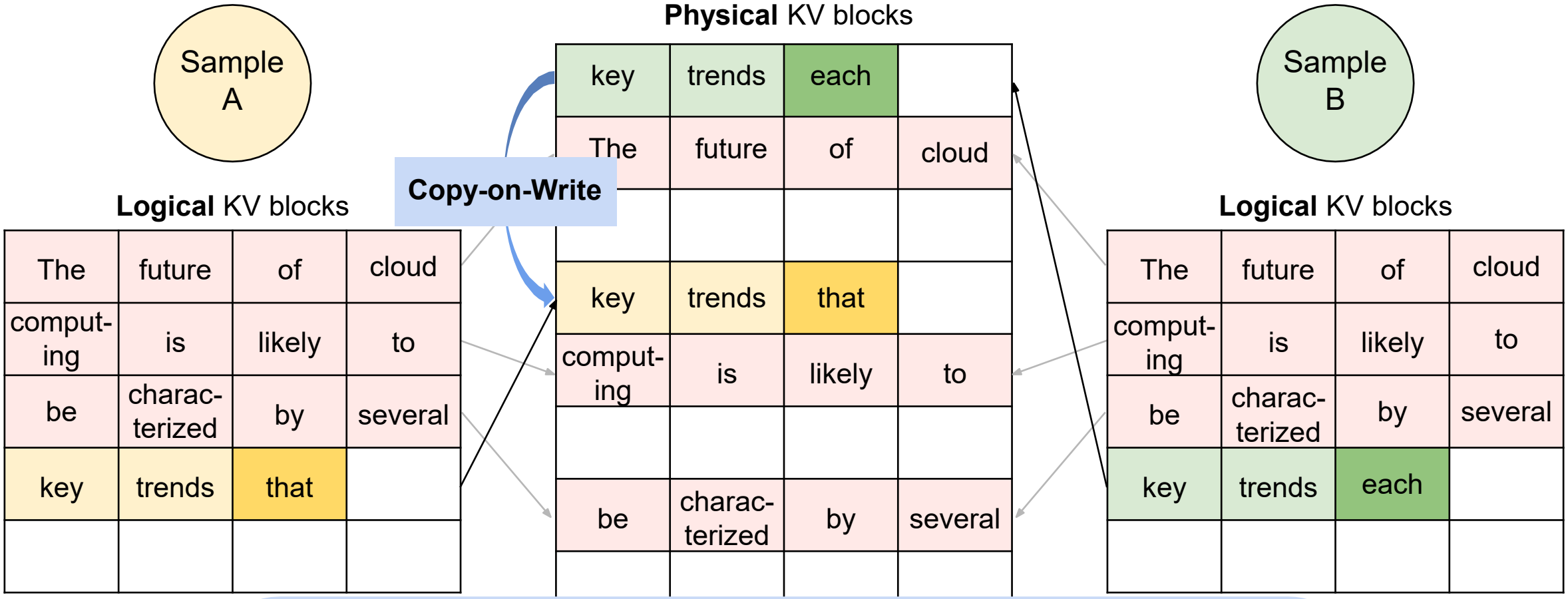
KV Cache Management with PagedAttention



Outline

- KV Cache Management with RadixAttention
- Cache-aware scheduling and load balancing
- Zero-overhead scheduler and worker

Waste in PagedAttention sharing



All prompt blocks (except the last) are *shared* across requests

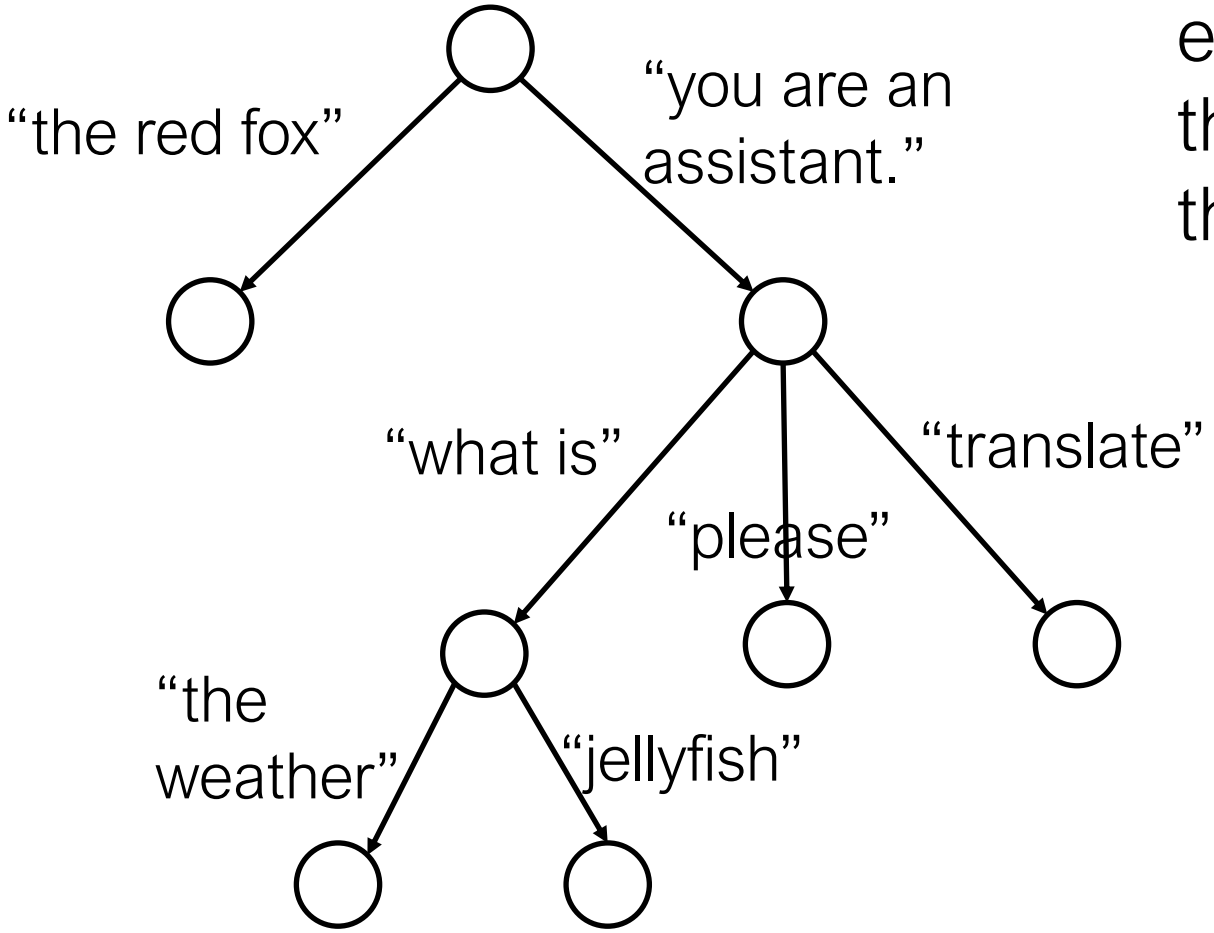
Managing KV Cache with RadixAttention

- KV mem pointers are stored in a radix tree (i.e. prefix tree)
  - each edge is a string
  - each node contains memory pointers of KVs
  - the whole path's string represent the prefix

- Retrieve KV cache for a prompt string
  - lookup a prompt string's longest prefix from the radix tree: very fast by matching the edge
  - returns: number of tokens in prefix matched, the matched node

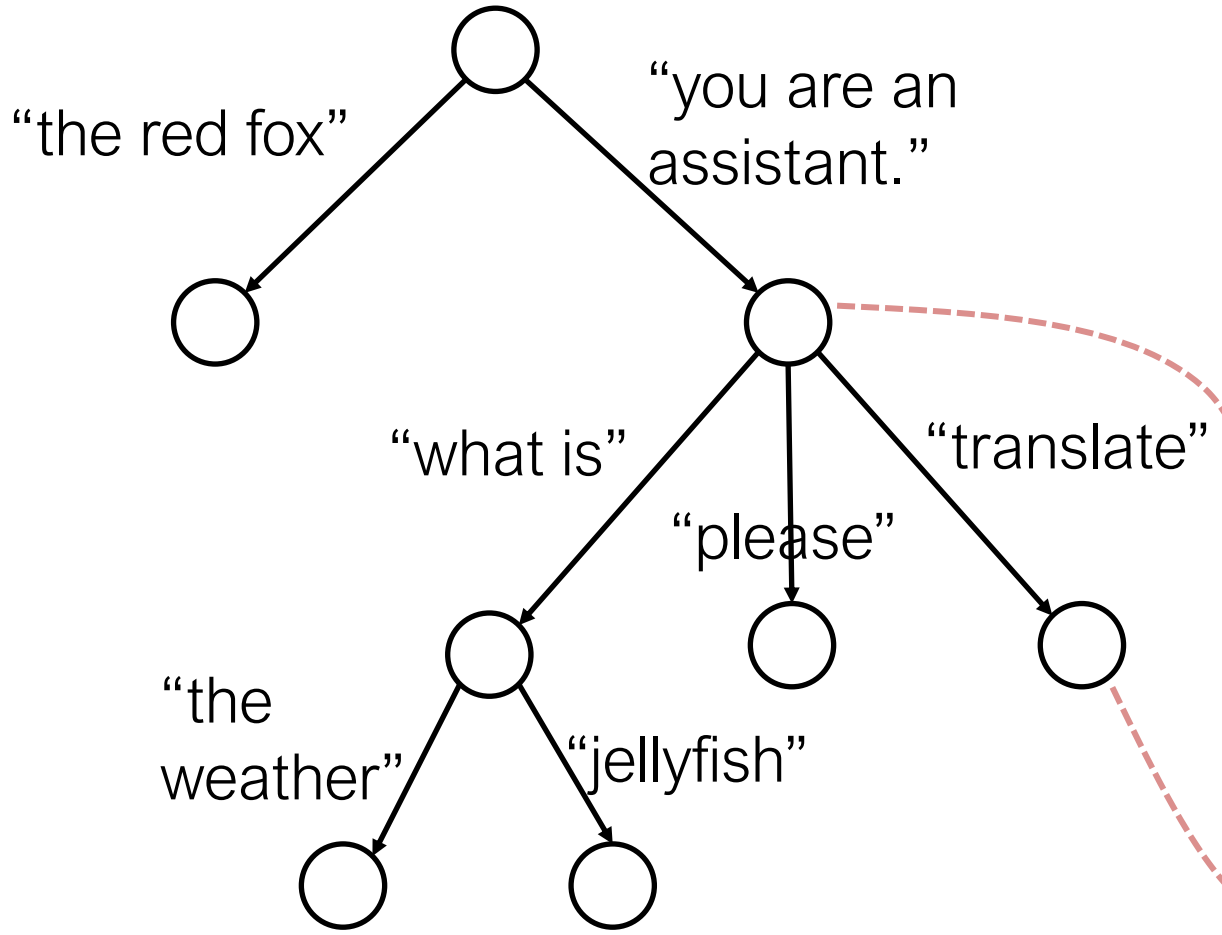
Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Radix Tree

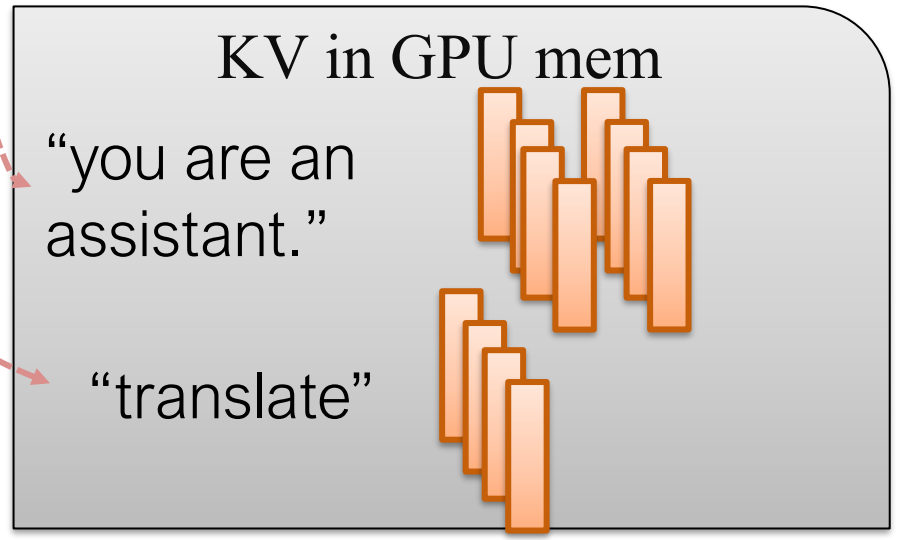


each edge is a string
the whole path’s string represent the prefix

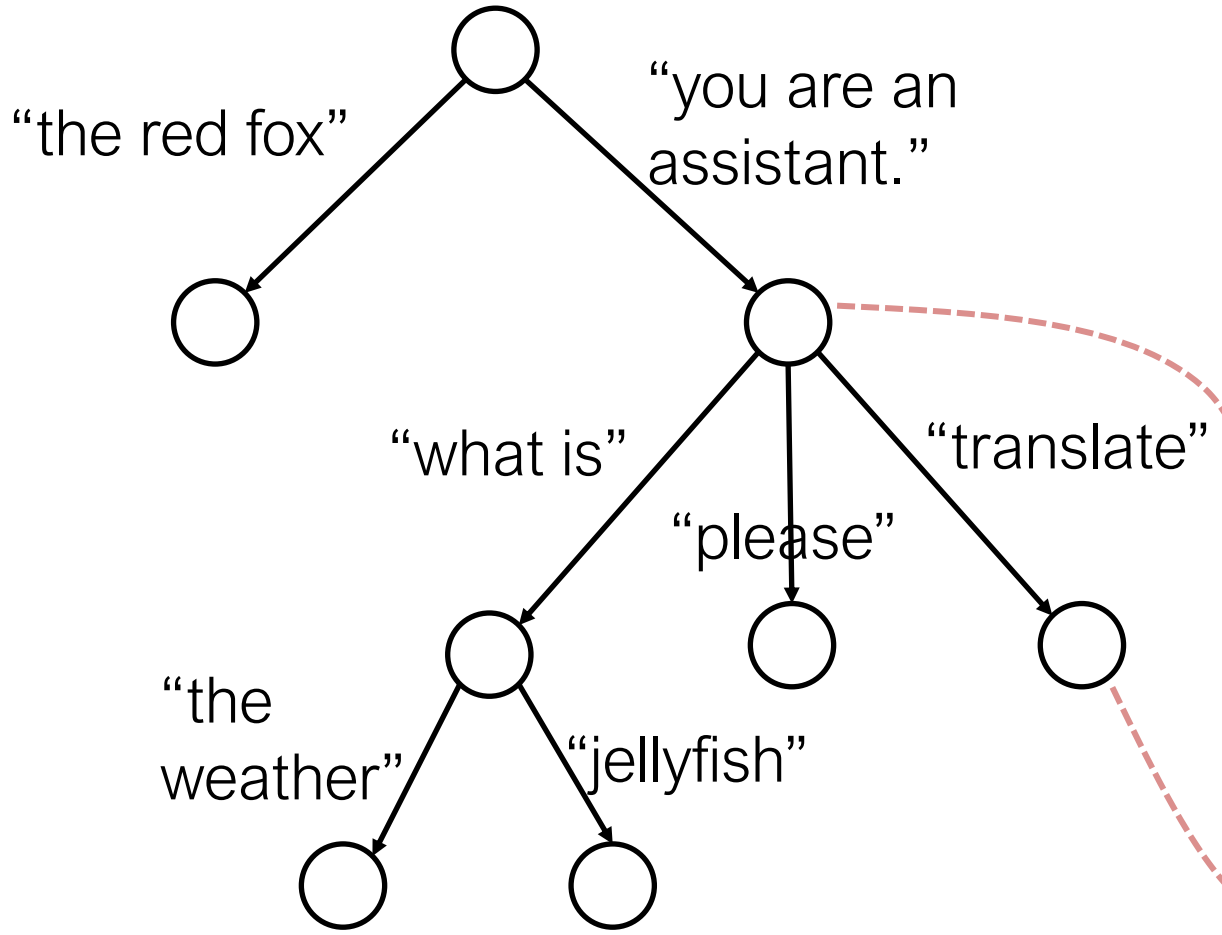
Radix Tree for KV Cache Management



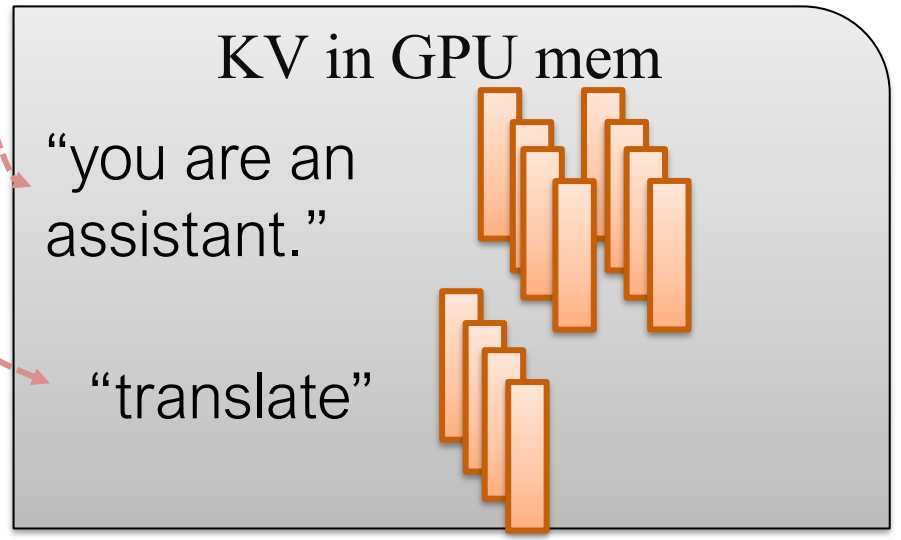
RadixTree is maintained in CPU
each edge is a string
the whole path’s string represent the prefix
node points to the GPU memory storing the KVs on the edge



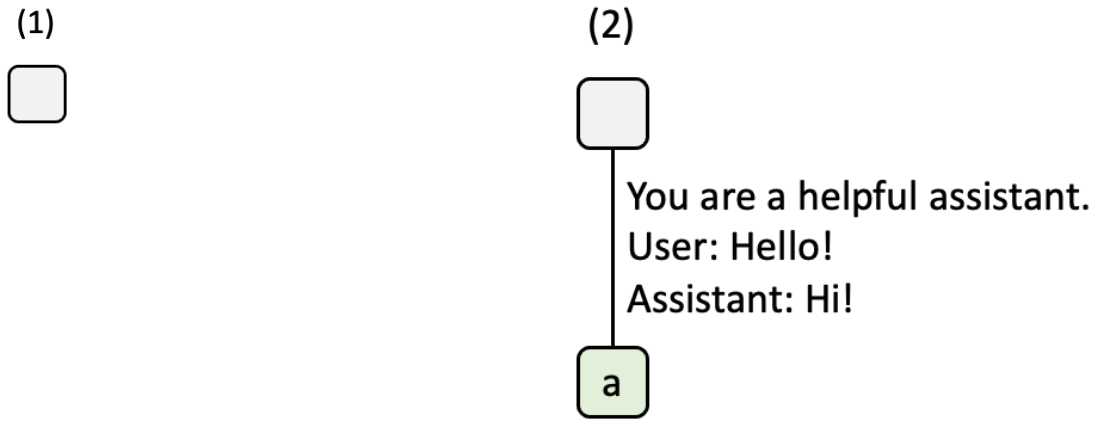
Radix Tree for KV Cache Management



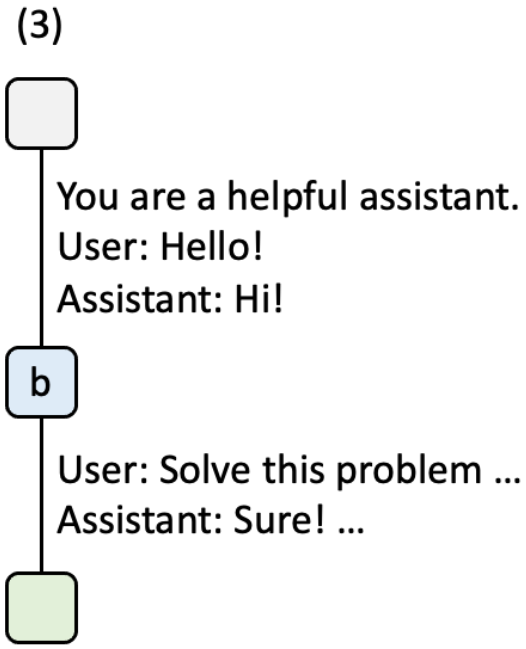
each edge is a string
the whole path’s string represent the prefix
node points to the GPU memory storing the KVs on the edge



Add KV for New Request Radix Tree

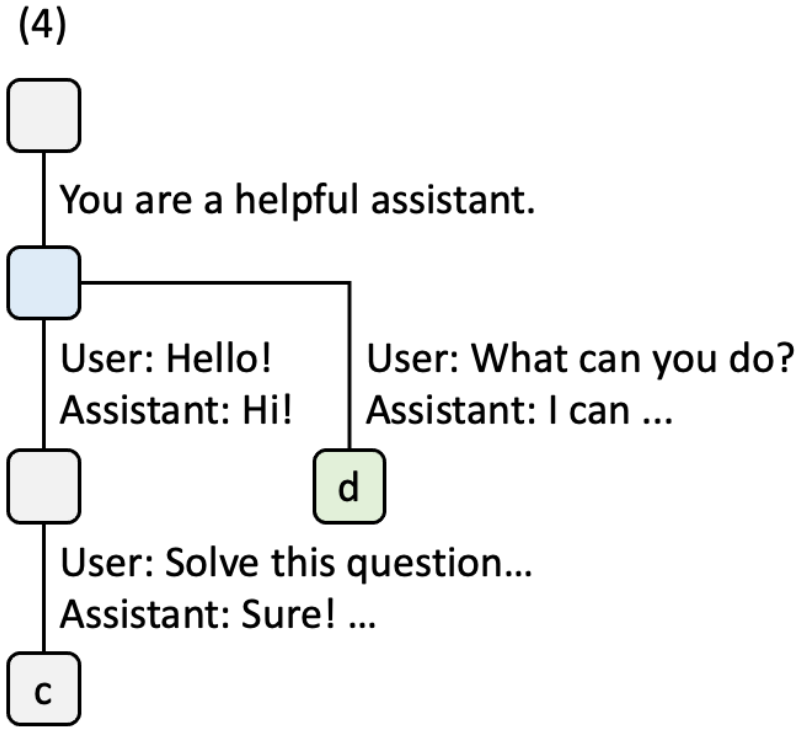


Add KVs for Next Round User Input in Chat



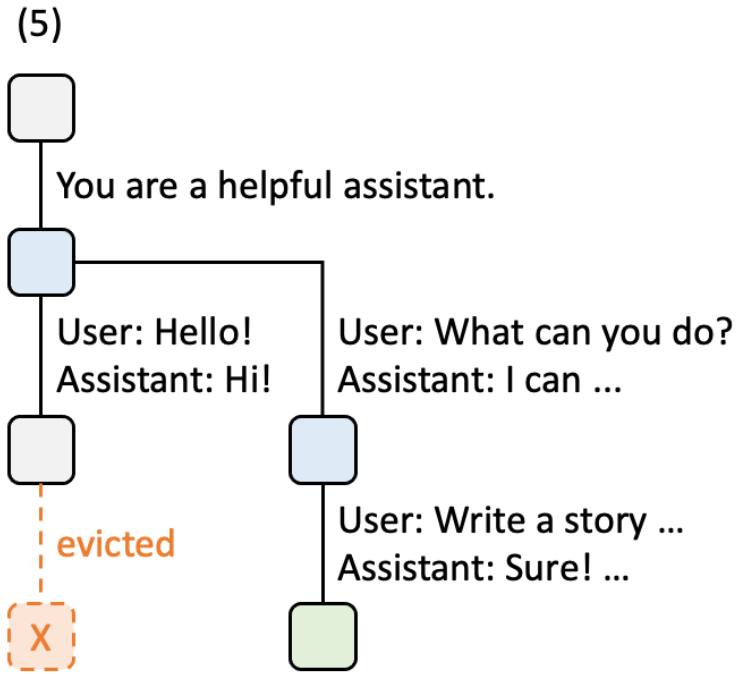
Node Split and Add for a new user's request

node split



Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

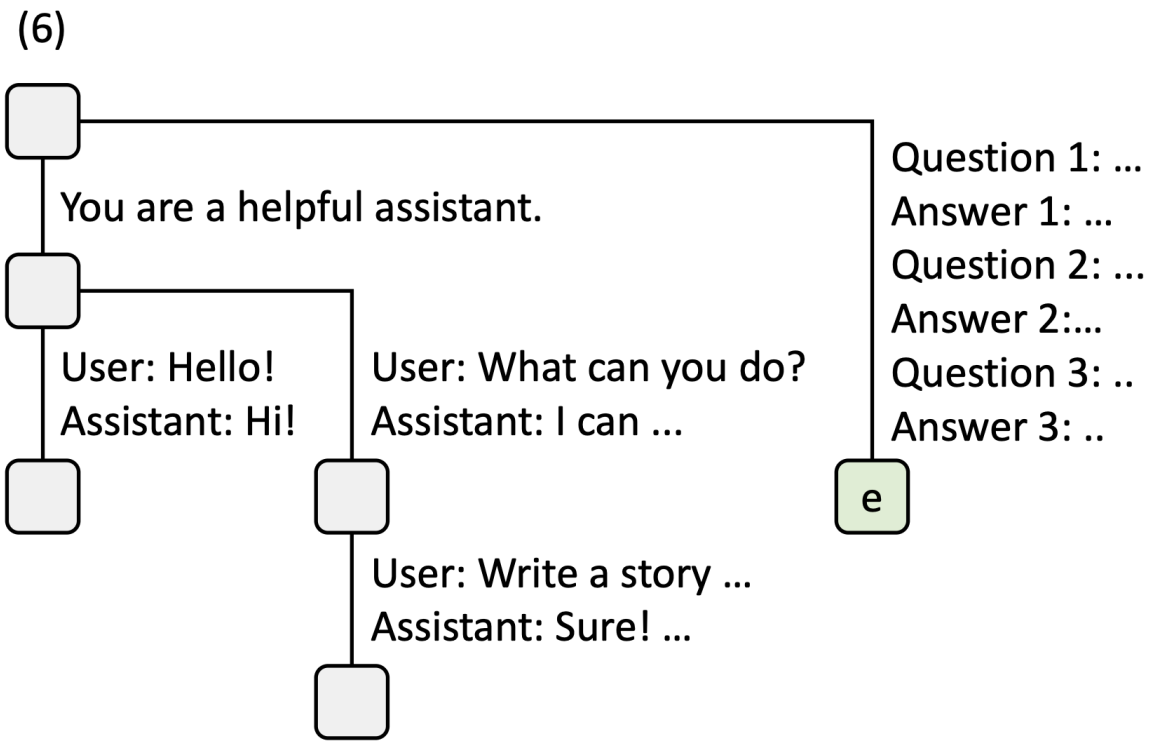
Node Eviction after next round of input



least used KV nodes are evicted from GPU Cache if over flow

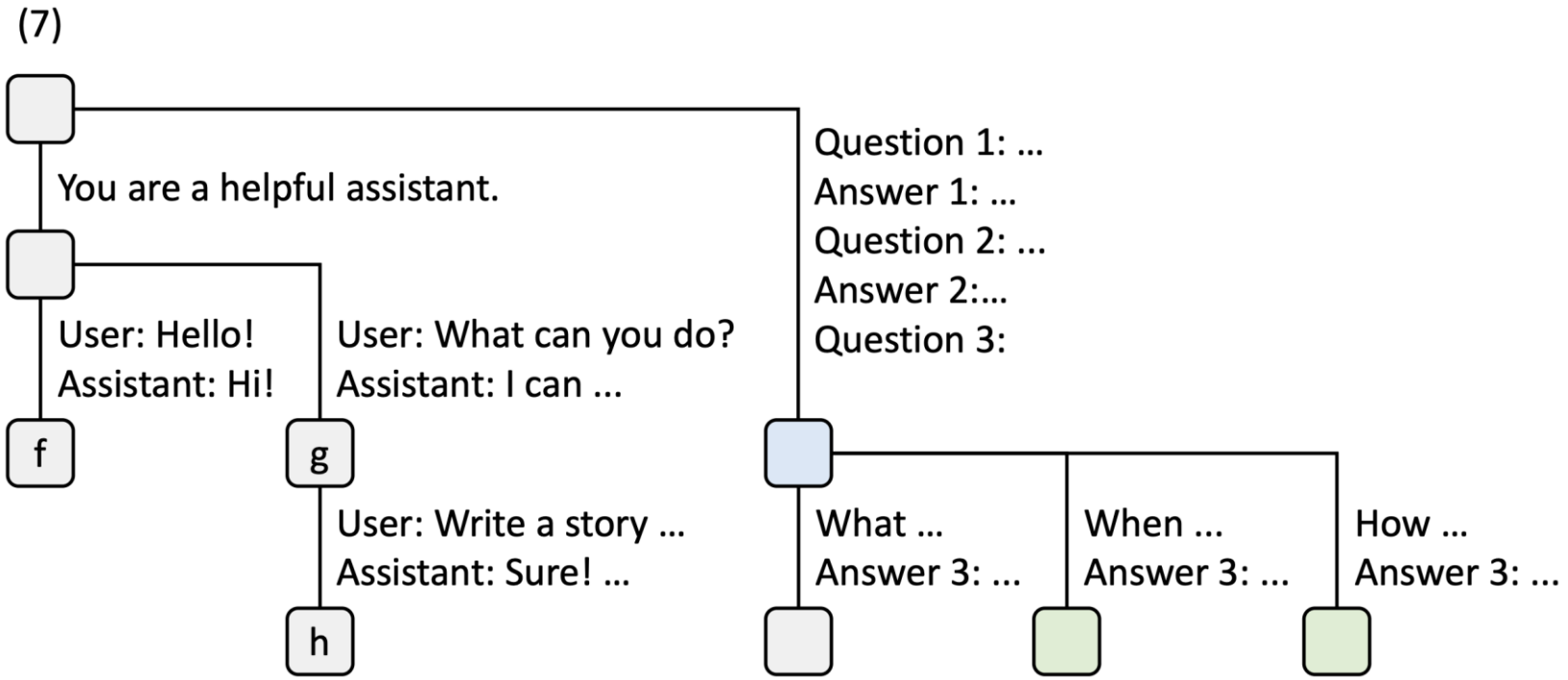
Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Add KV for User Request with In-context Examples



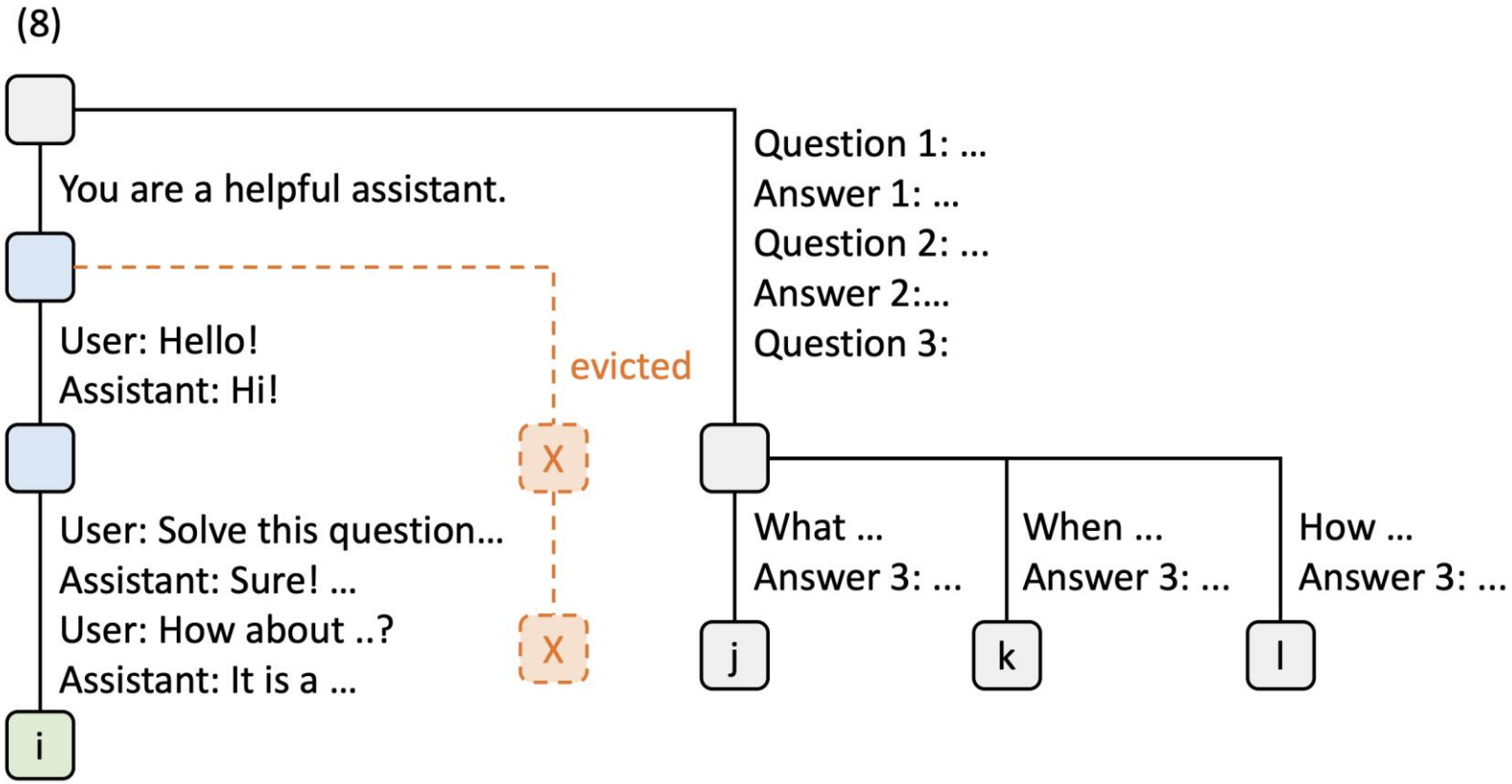
Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Add KV for User Request with In-context Examples



Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

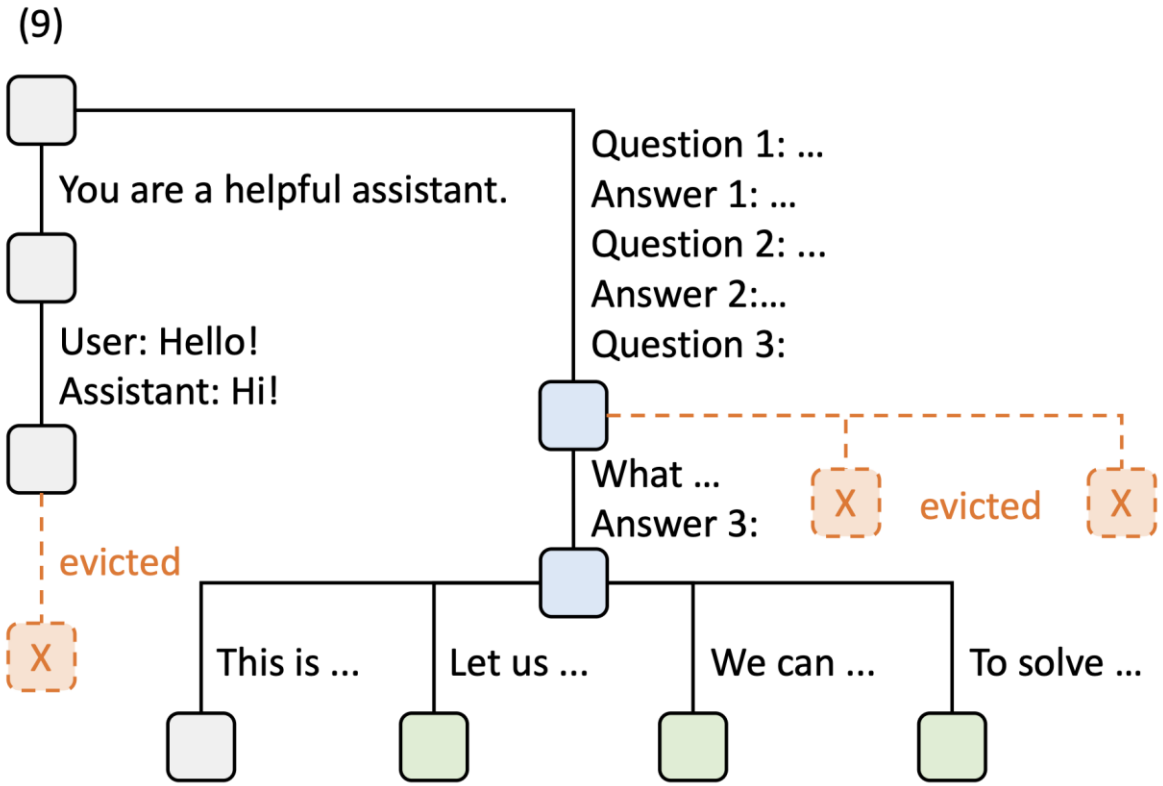
KV Eviction after next round of prior user session



least used KV nodes are evicted from GPU Cache if over flow

Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Next round of In-context Prompt – Evicting KVs



least used KV nodes are evicted from GPU Cache if over flow

Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Outline

- KV Cache Management with RadixAttention
- Cache-aware scheduling and load balancing
- Zero-overhead scheduler and worker

Scheduler with RadixAttention

1. Receive input messages
2. Stream model outputs
3. Check the stop conditions
4. Maintaining radix tree and request reorder
5. Allocate memory for the next batch

```
while True:
    recv_reqs = recv_requests()
    process_input_requests(recv_reqs)
    batch = get_next_batch_to_run()
    result = run_batch(batch)
    process_batch_result(batch, result)
```

Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Inside Scheduler of SGLang

- get_next_batch_to_run()
  - if finishing prefill: moving the completed prefill to decode batch
  - if new prefill request: get_new_batch_prefill
    - takes a new request, depending on the available spot in batch, and waiting queue of requests (calculating the priority of requests), then addPrefill
  - if no new prefill: continue the decode update_running_batch()
    - check_decode_mem: is there enough GPU memory for running
      - yes ➔ prepare_for_decode, reduce new_token_ratio
      - no => batch.retract_decode, increase new_token_ratio, put the req back to prefill_waiting_list
  - process_batch_result: check if any req is complete. free the kv cache (only reference, not actually free mem)

KV Cache-aware Scheduler with RadixAttention

- For requests in the queue, sort the requests according to matched prefix length (to maximize the KV cache hit)
- Hit rate: $\frac{\#\text{cached tokens}}{\#\text{total prompt tokens}}$

```
def get_next_batch():
    # Match prefix
    for req in waiting_queue:
        req.prefix_length  = match_prefil(req, radix_tree_cache)

    # Sort according to the prefix_length
    waiting_queue.sort()

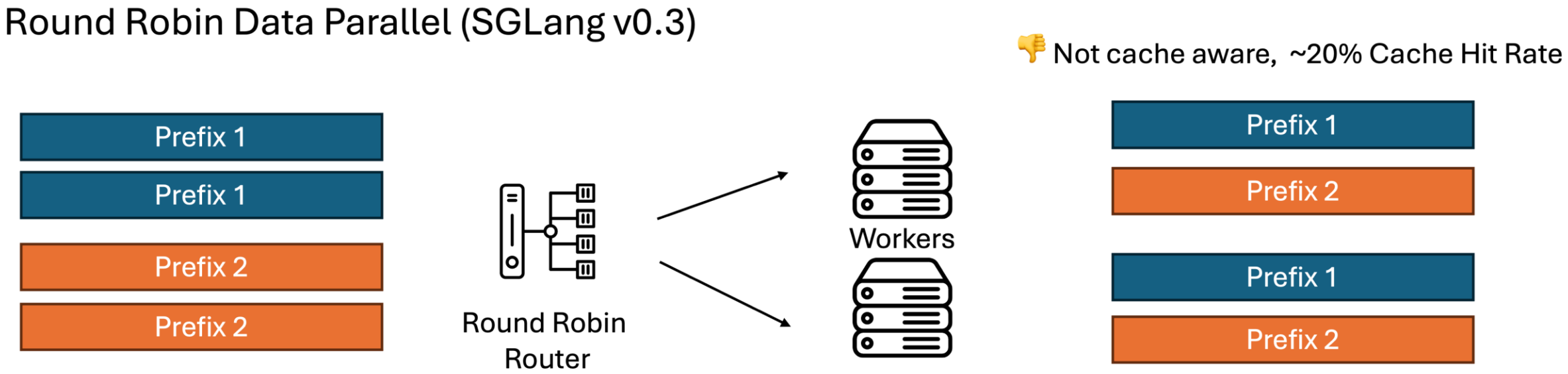
    # Add requests in the next batch within memory constraint
    next_prefill_batch = []
    for req in waiting_queue:
        if can_run(req):
            next_prefill_batch.append(req)

    return next_prefill_batch
```

Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Cache-aware Load Balancer

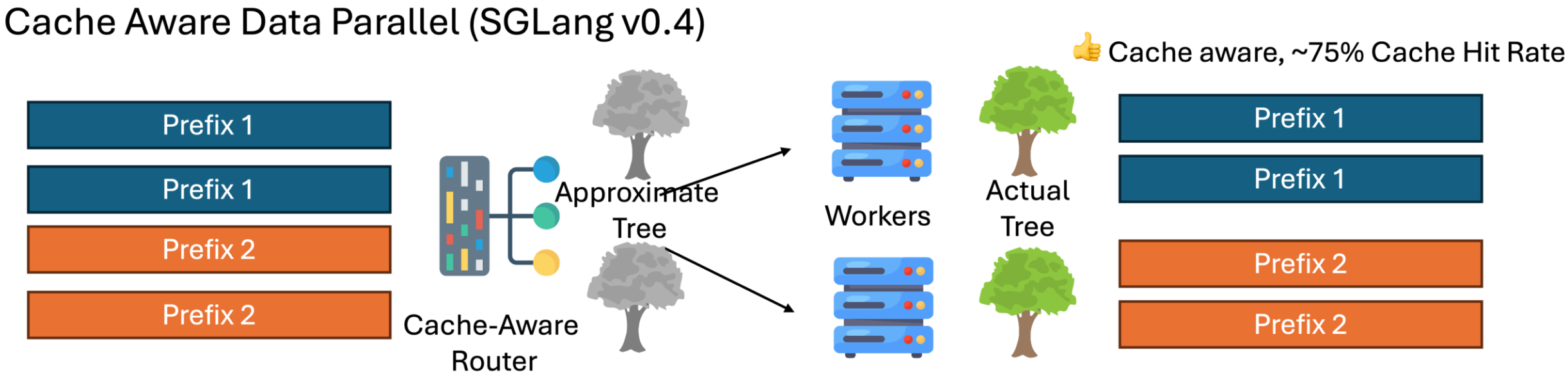
- Maintain KV-Caches on multiple worker nodes



Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

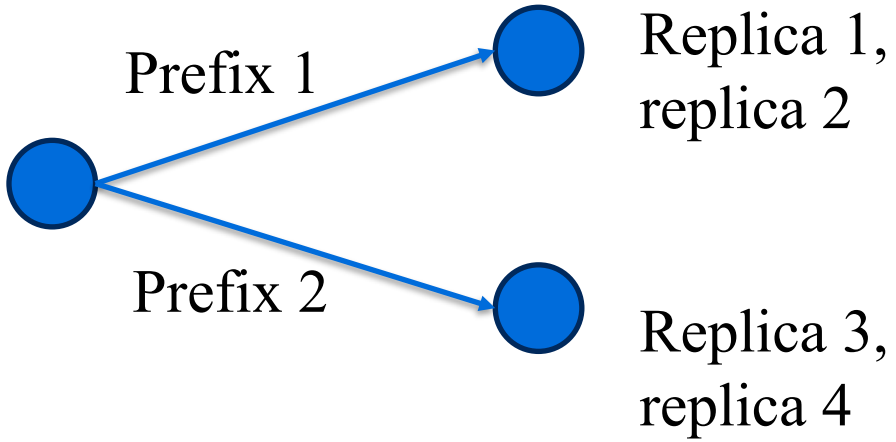
Cache-aware Load Balancer

- Predict prefix KV cache hit rates on workers
- Select those with the highest match rates.



Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Cache-aware Load Balancer



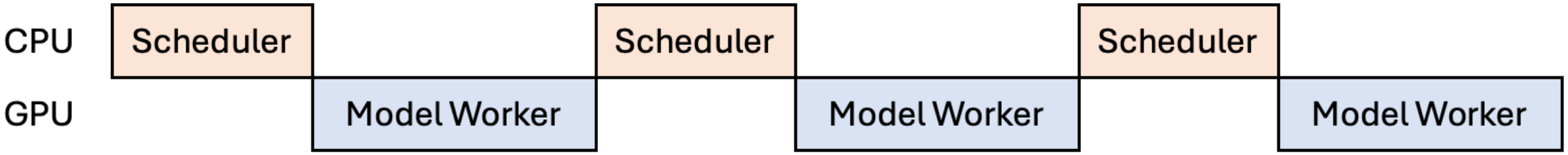
	Round robin	Cache aware load balancer
Throughput (token/s)	82665	158596
Cache hit rate	20%	75%

Zheng et al. SGLang: efficient execution of structured language model programs. NeurIPS 2024.

Outline

- KV Cache Management with RadixAttention
- Cache-aware scheduling and load balancing
- Zero-overhead scheduler and worker

CPU Overhead with Scheduler



Ideal case: full GPU usage, eliminate CPU overhead

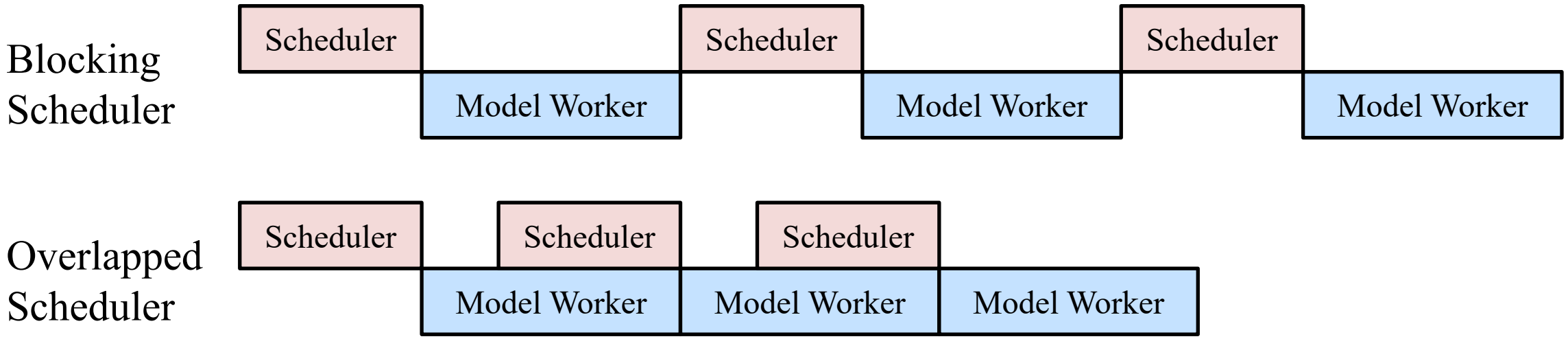
Jobs of the CPU scheduler

- Receives input messages from the user
- Processes results from the model worker
- Checks the stop conditions
- Runs prefix matching and request reorder
- Allocates memory for the next batch

Pseudo code (every line is blocking)

```
while True:
    recv_reqs = recv_requests()
    process_input_requests(recv_reqs)
    batch = get_next_batch_to_run()
    result = run_batch(batch)
    process_batch_result(batch , result)
```

Overlap CPU scheduler and GPU worker



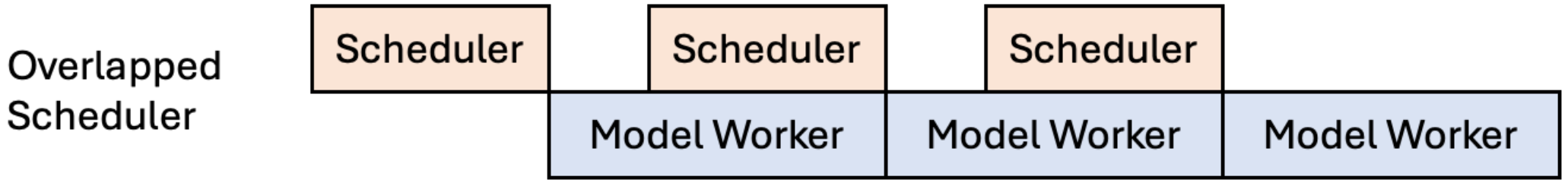
Jobs of the CPU scheduler

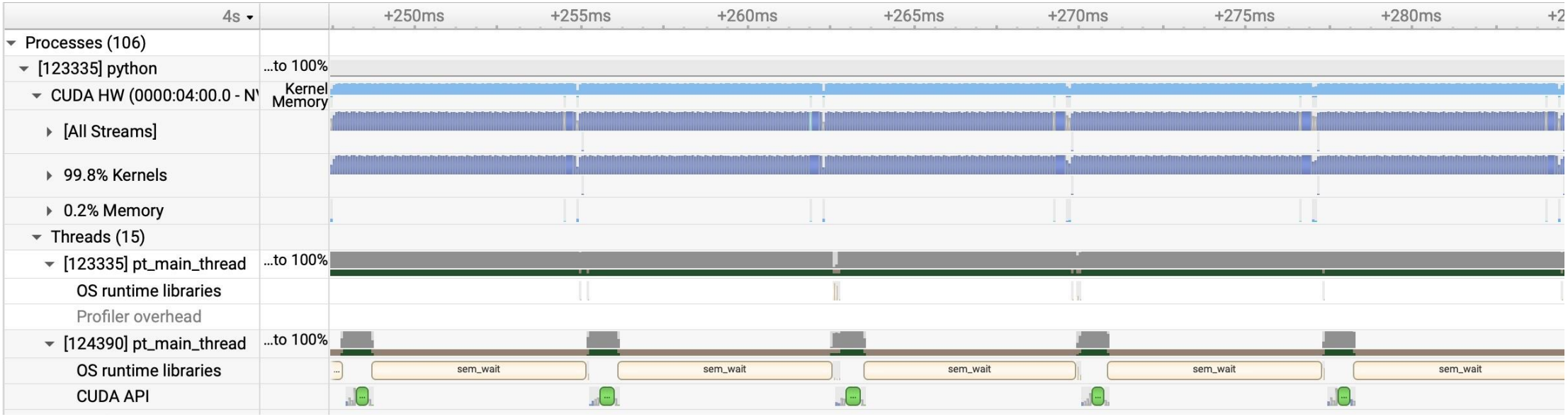
- Receive input messages
- Stream model outputs
- Check the stop conditions
- Maintaining radix tree and run prefix matching
- Allocate memory for the next batch

Ideas

- Resolve the dependency by delaying the stop condition check
- Use CUDA events and streams to do fine-grained scheduling

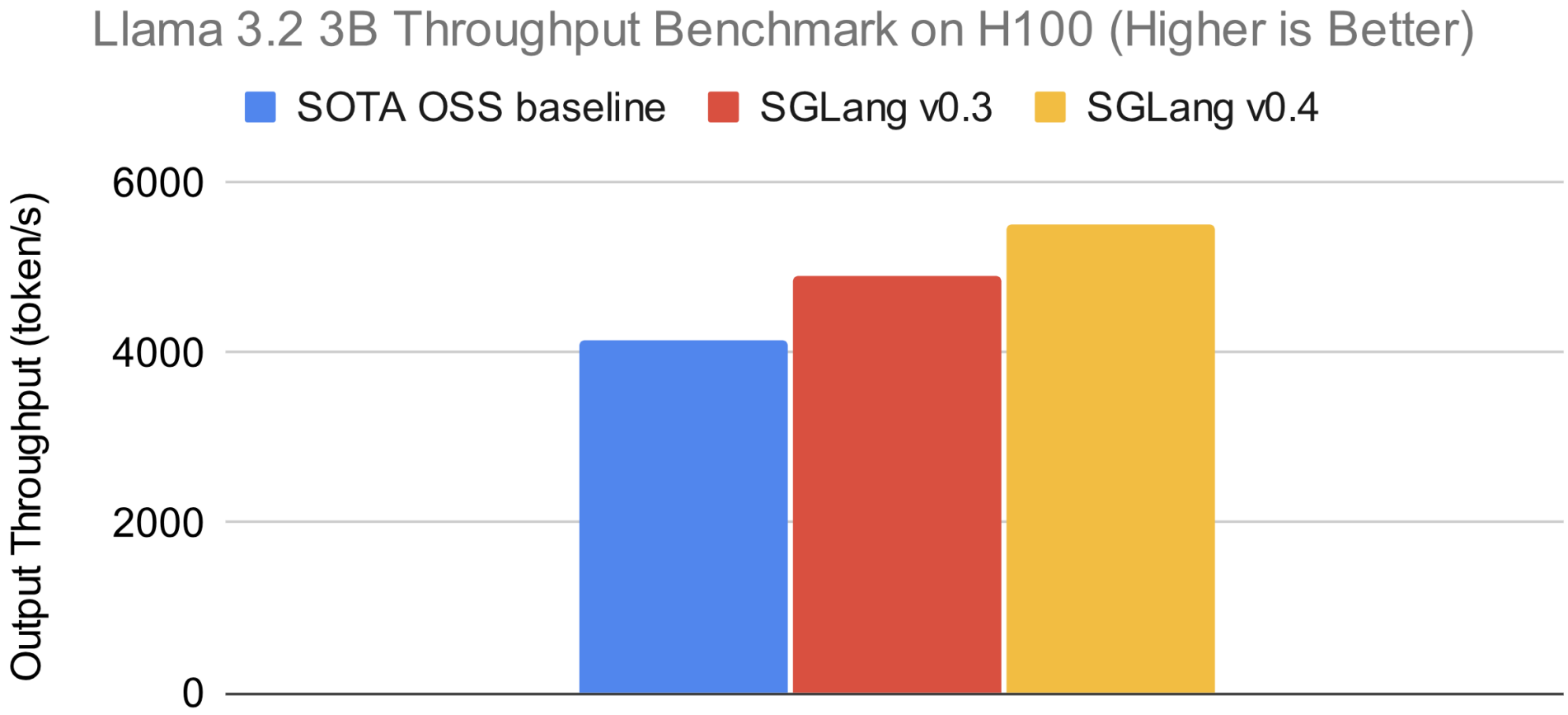
Zero-overhead CPU scheduling



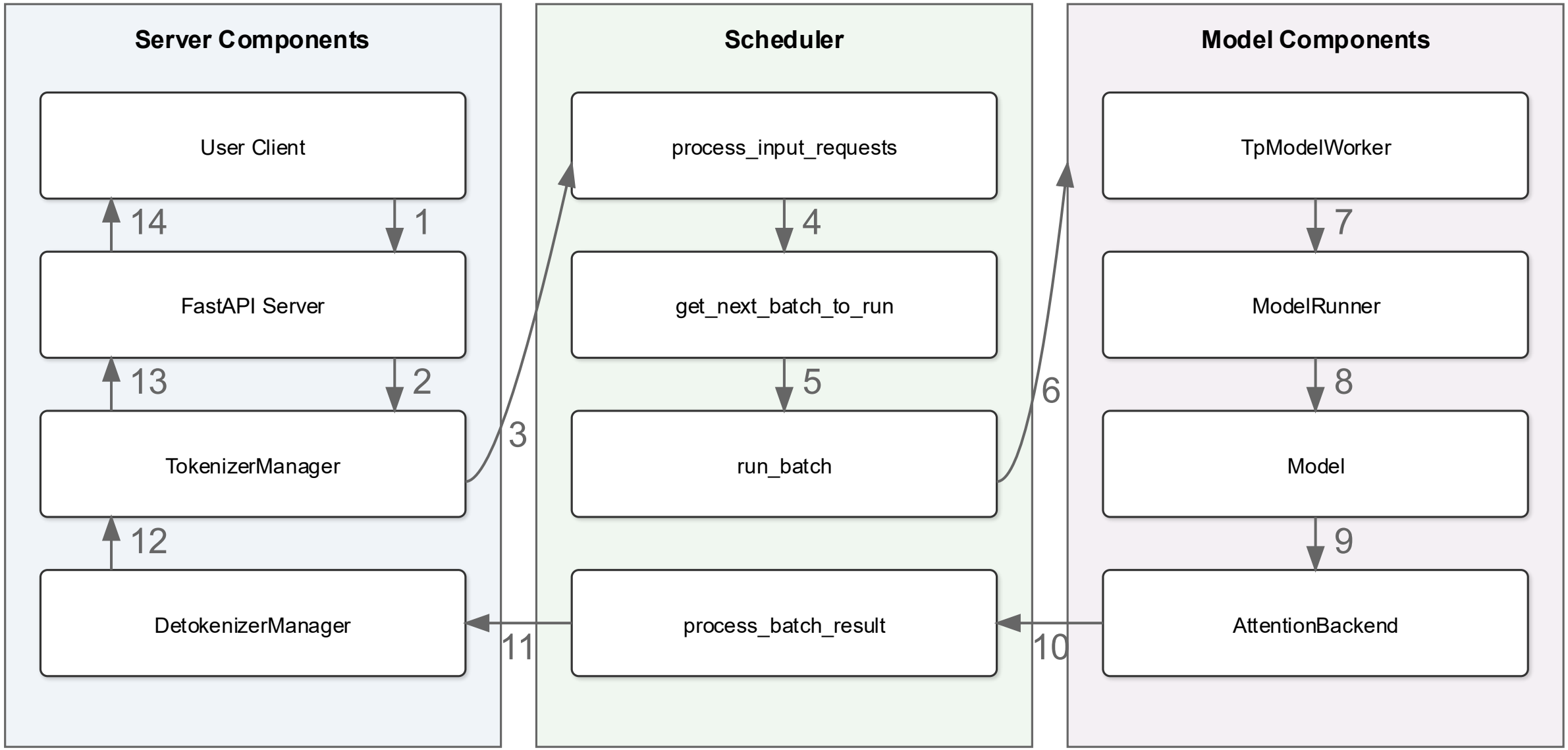


Increased Performance of Scheduler-worker Overlapping

1.3x faster than SOTA OSS baseline



Full Execution Flow on SGLang



Additional Features of LLM Server

1. Speculative Decoding (SpecForge)
2. Efficient Constraint Decoding (XGrammar)
3. Expert Parallelism: DeepEP
4. Model-specific Support: MLA

Summary of LLM Serving

1. Basic LLM Request Scheduler

2. Continuous Batching
   - handling requests with varying lengths

3. KV Cache Management with PagedAttention
   - Use block table to organize the physical attention block

4. KV Cache Management with RadixAttention
   - using radix tree to maintain KVs based on prompt prefix
   - Load balancing

Quiz