LLM Sys

Design of Efficient LLM Inference Server

Lei Li

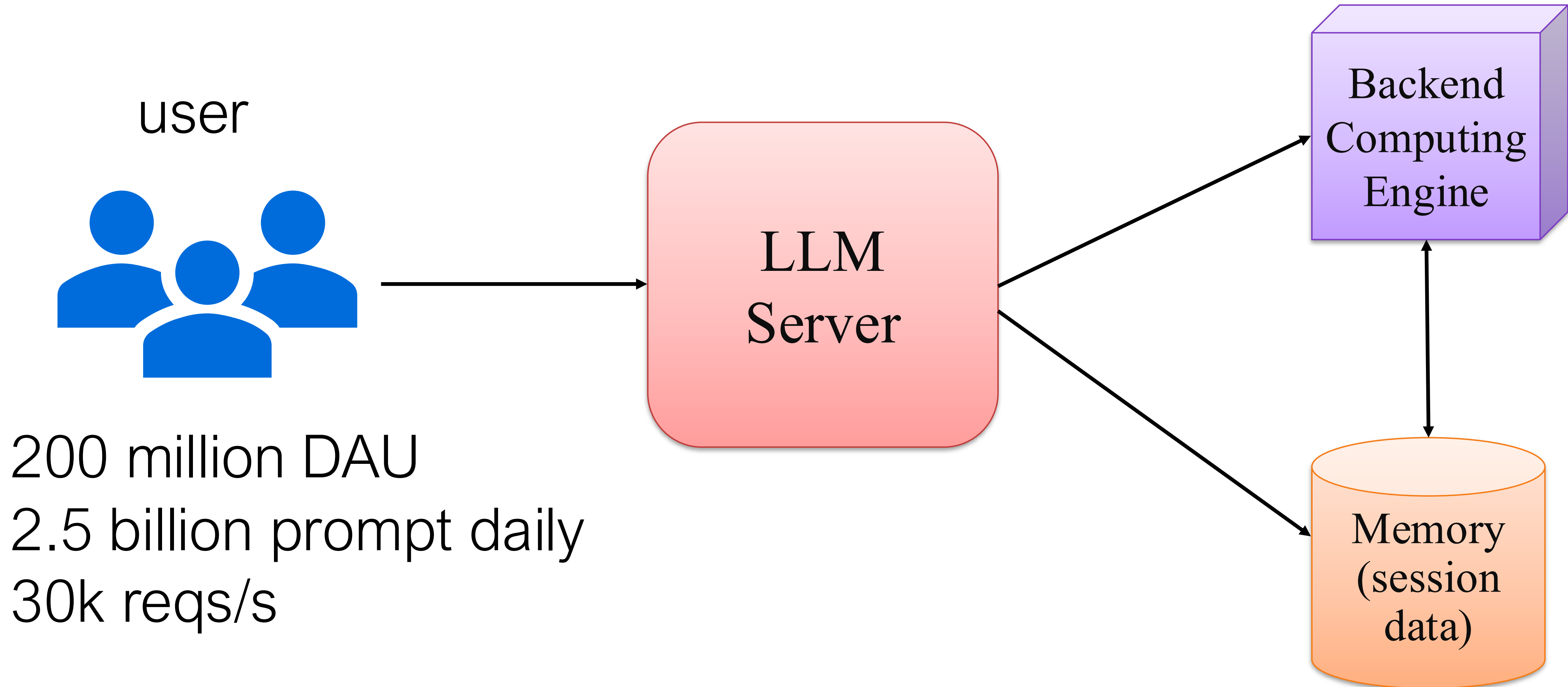
Language Technologies Institute

Carnegie Mellon University
School of Computer Science

Outline

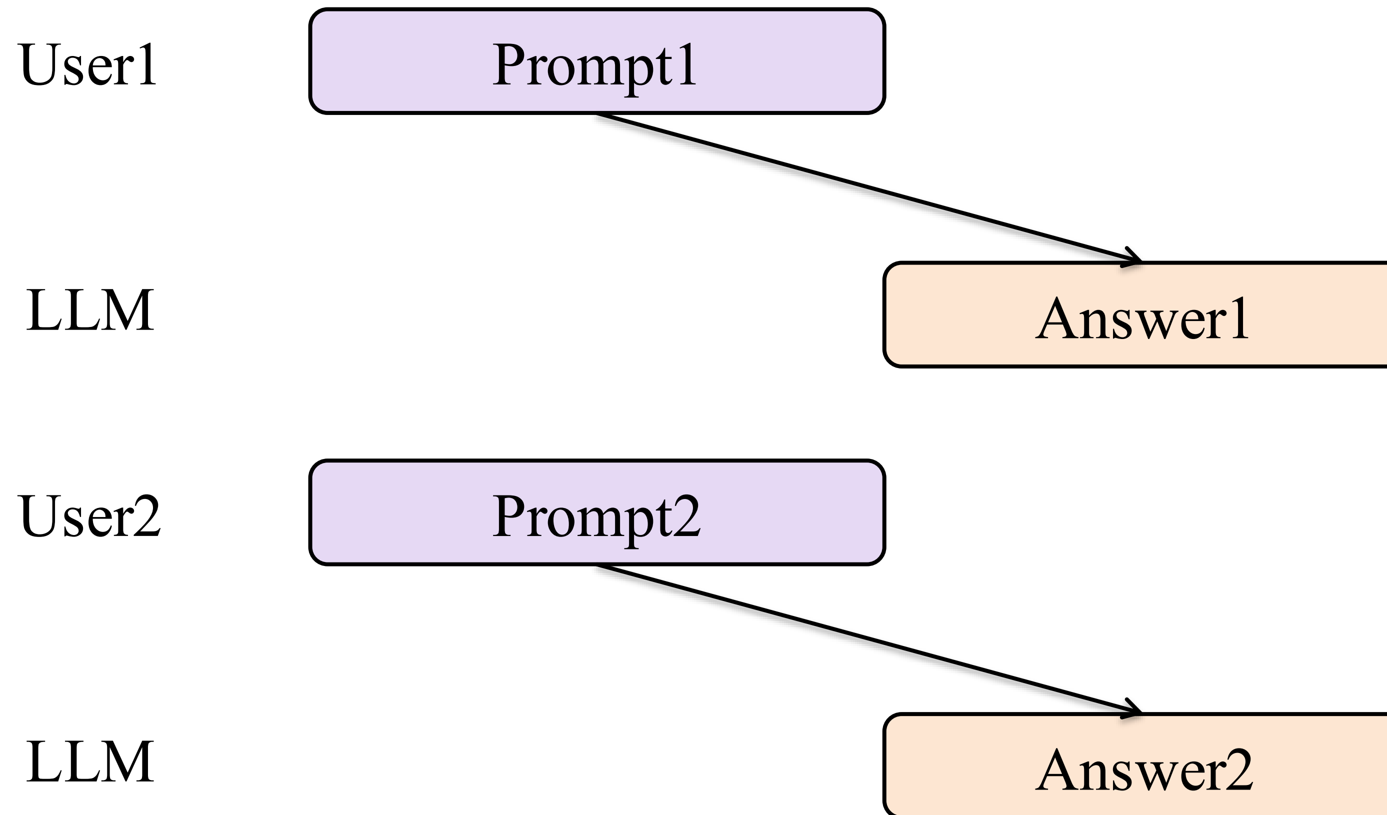
- Overview of LLM Inference Server
- LLM Request Scheduler
- Scheduler with Continuous Batching
- KV Cache Management with RadixAttention
- Cache-aware scheduling and load balancing
- Zero-overhead scheduler and worker

LLM Serving



LLM Serving Pattern

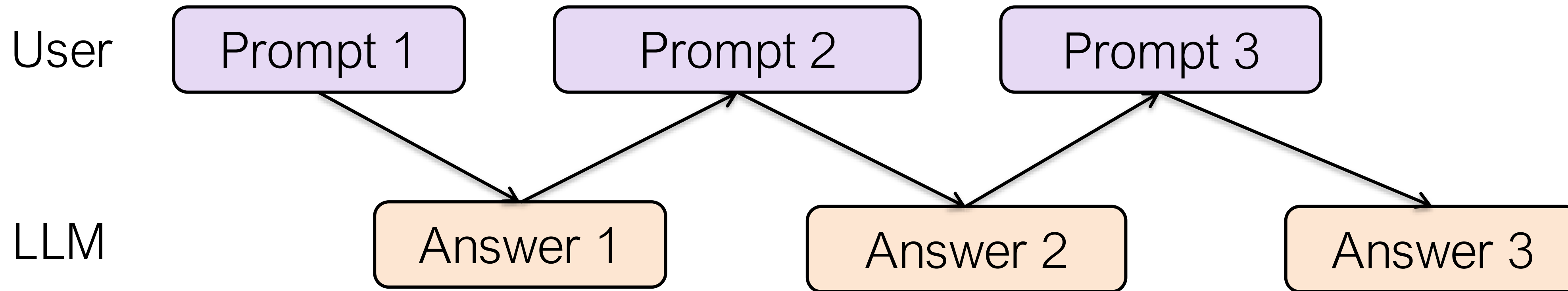
Multi-user Single Round Generation



Different users may issue same prompts

LLM Serving Pattern

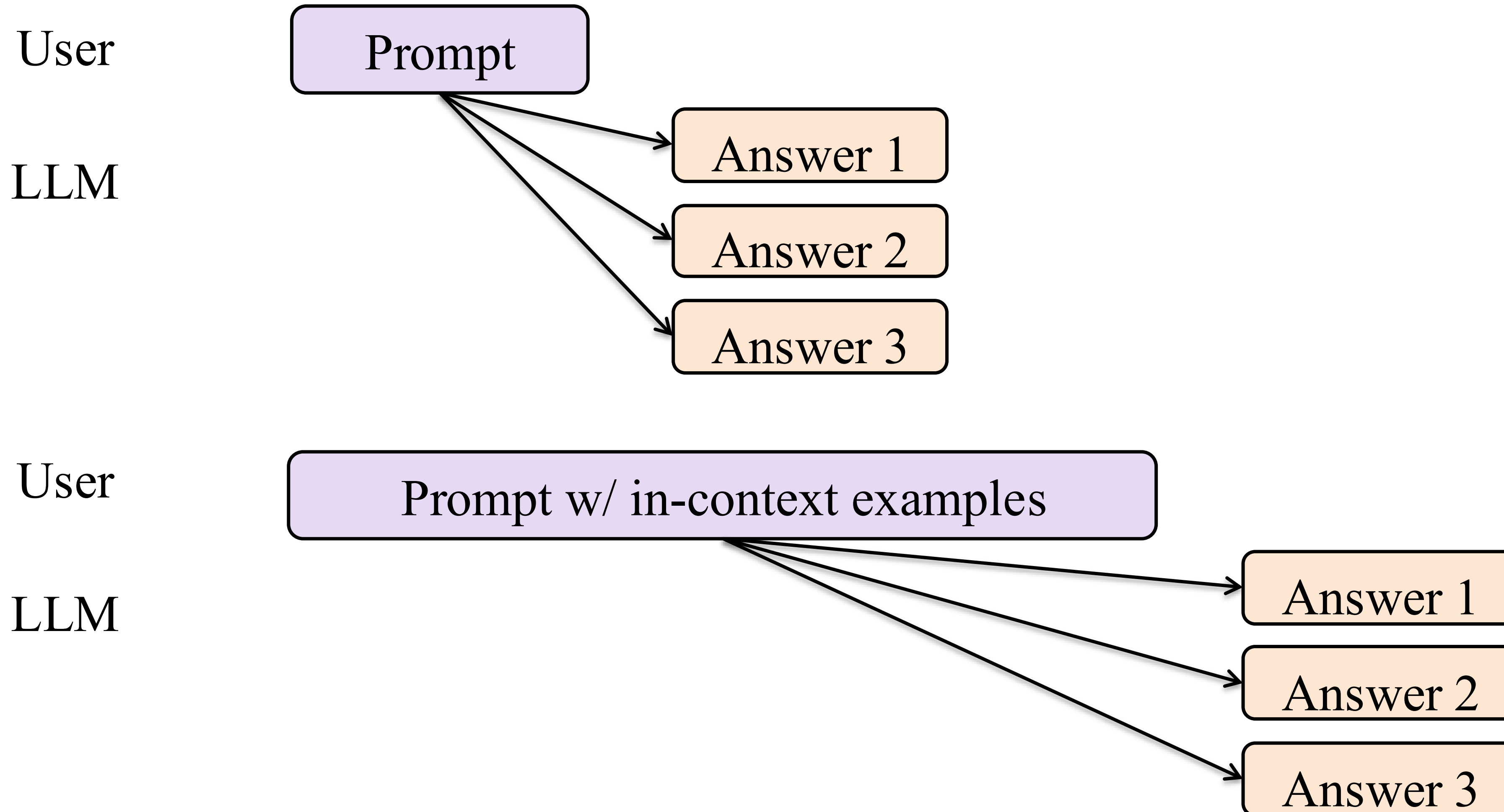
Multi-round Chat



Should LLM server keep the user chat history in GPU memory and wait for next turn?

LLM Serving Pattern

Multi-Generation



Execute the LLM generation three times?

Design Goals of Efficient LLM Server

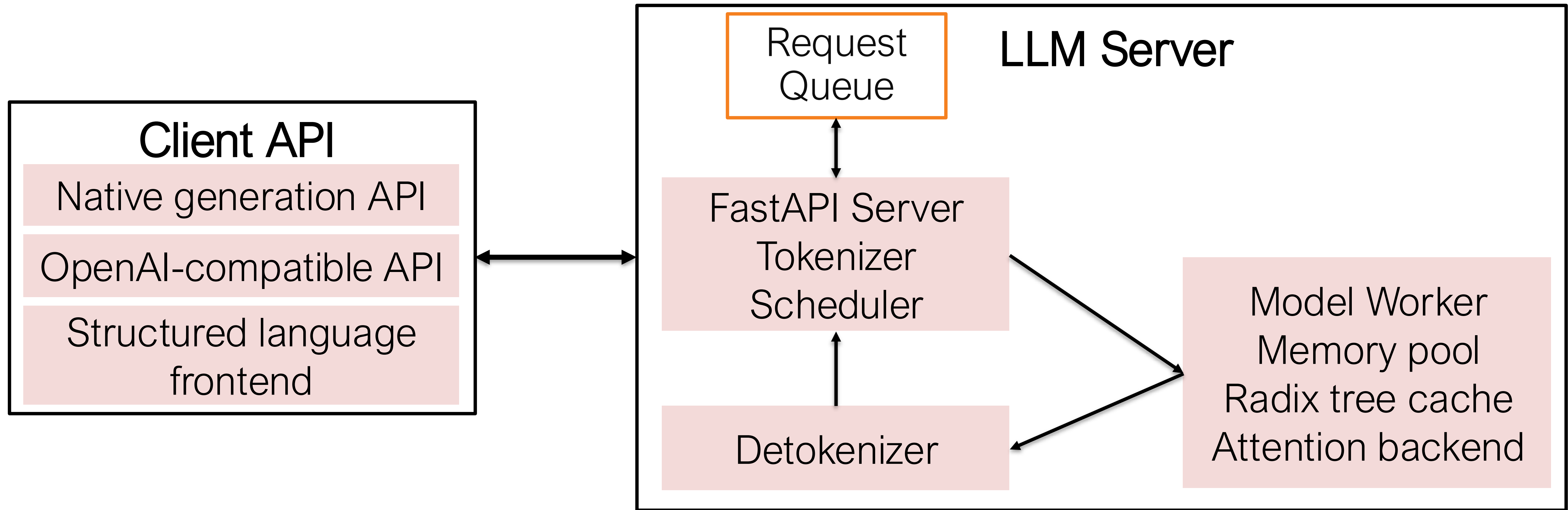
- Maintain multi-user chat sessions
- Handle massive user requests at one time
  - varying sequence generation lengths
  - requests with common prompt prefix
- Optimizing for Throughput and Latency
- Utilize heterogeneous devices (CPU/GPU)
- Examples: ORCA, SGLang, vLLM

Outline

- Overview of LLM Inference Server
- LLM Request Scheduler
- Scheduler with Continuous Batching
- KV Cache Management with RadixAttention
- Cache-aware scheduling and load balancing
- Zero-overhead scheduler and worker

LLM Server Architecture

SGLang / vLLM share similar arch



Lightweight and customizable codebase in Python/PyTorch

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)
```

Server scheduler example

```python
while True:
    recv_reqs = self.recv_requests()
    self.process_input_requests(recv_reqs)

    batch = self.get_next_batch_to_run()
    self.cur_batch = batch

    if batch:
        result = self.run_batch(batch)
        self.result_queue.append((batch.copy(), result))

        if self.last_batch is None:
            # Create a dummy first batch to start the pipeline for overlap schedule.
            # It is now used for triggering the sampling_info_done event.
            tmp_batch = ScheduleBatch(
                reqs=None,
                forward_mode=ForwardMode.DUMMY_FIRST,
                next_batch_sampling_info=self.tp_worker.cur_sampling_info,
            )
            self.process_batch_result(tmp_batch, None)

    if self.last_batch:
        # Process the results of the last batch
        tmp_batch, tmp_result = self.result_queue.popleft()
        tmp_batch.next_batch_sampling_info = (
            self.tp_worker.cur_sampling_info if batch else None
        )
        self.process_batch_result(tmp_batch, tmp_result)
    elif batch is None:
        # When the server is idle, do self-check and re-init some states
        self.check_memory()
        self.new_token_ratio = self.init_new_token_ratio

    self.last_batch = batch
```

Outline

- Overview of LLM Inference Server
- LLM Request Scheduler
- Scheduler with Continuous Batching
- KV Cache Management with RadixAttention
- Cache-aware scheduling and load balancing
- Zero-overhead scheduler and worker

Serving Challenge: Waiting for Longest Gen

- A Request Batch may lead to different lengths.
- Naïve implementation needs to wait for the longest seq

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1				
S_2	S_2	S_2					
S_3	S_3	S_3	S_3				
S_4	S_4	S_4	S_4	S_4			

prefill decode

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1	S_1	END		
S_2	S_2	S_2	S_2	S_2	S_2	S_2	END
S_3	S_3	S_3	S_3	END			
S_4	S_4	S_4	S_4	S_4	S_4	END	

prefill decode

Serving with Continuous Batching

- Iteration-level: at each iteration (i.e. each token generation step) schedule new request whenever one request finishes in a batch

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1				
S_2	S_2	S_2					
S_3	S_3	S_3	S_3				
S_4	S_4	S_4	S_4	S_4			

prefill decode

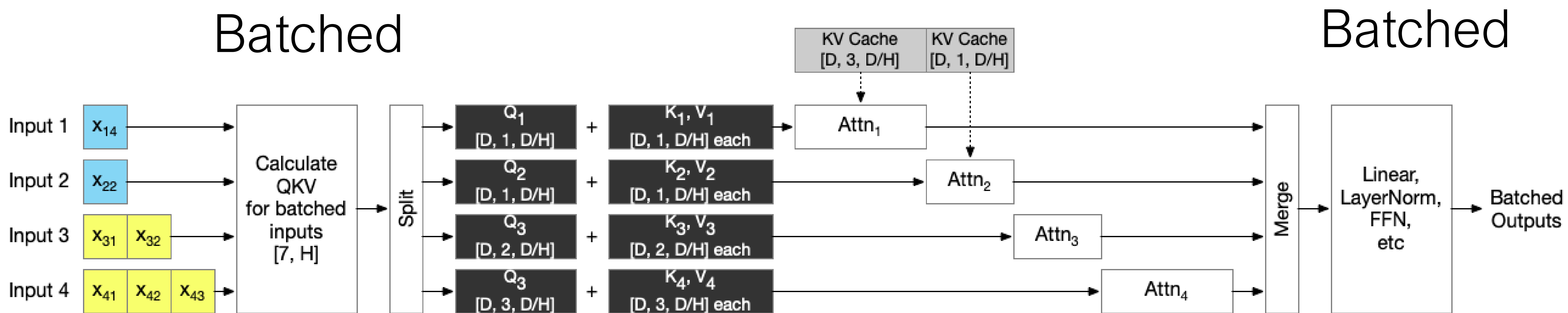
T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
S_1	S_1	S_1	S_1	S_1	END	S_6	S_6
S_2	S_2	S_2	S_2	S_2	S_2	S_2	END
S_3	S_3	S_3	S_3	END	S_5	S_5	S_5
S_4	S_4	S_4	S_4	S_4	S_4	END	S_7

prefill decode prefill

ORCA: A Distributed Serving System for Transformer-Based Generative Models. Yu et al. OSDI 2022

Serving with Selective Batching

- Selectively batch all non-attention operations (linear, layer norm, GeLU, etc)



prefill is grouped into one step

Executed by Attention Engine (e.g. PagedAttention)

ORCA: A Distributed Serving System for Transformer-Based Generative Models. Yu et al. OSDI 2022

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)

Outline

- Overview of LLM Inference Server
- LLM Request Scheduler
- Scheduler with Continuous Batching
- KV Cache Management with RadixAttention
- Cache-aware scheduling and load balancing
- Zero-overhead scheduler and worker

KV Cache Management

- Standard NN inference only stores the current layer's output, but do not store all intermediate results for each layer
- This is ok for LSTM/RNN, but it is increasingly slower for Transformer inference – generating each next token would require all previous keys and values for each Transformer layer.
- Store all keys, values for each layer in GPU memory ➔ KV Cache

KV Cache Management

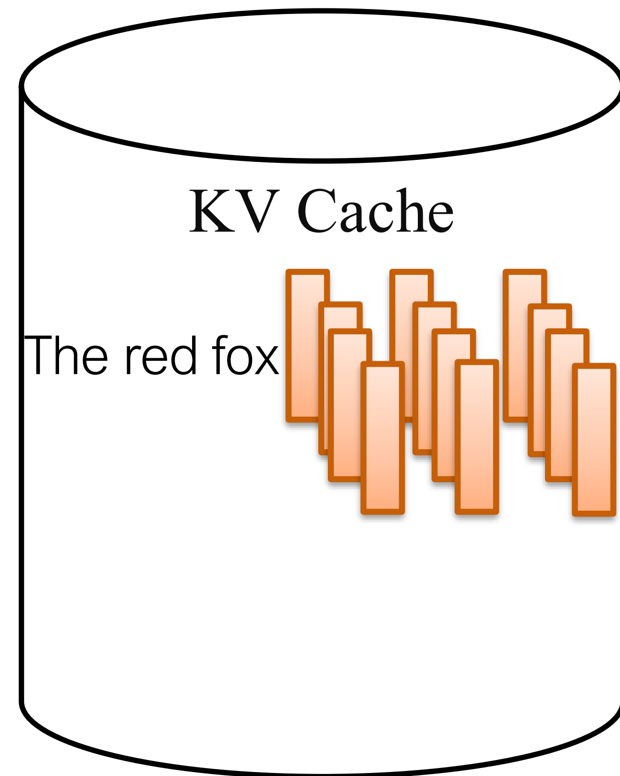
Layer2

Q K V Q K V Q K V Q K V

Layer1

Q K V Q K V Q K V Q K V

The red fox sits



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

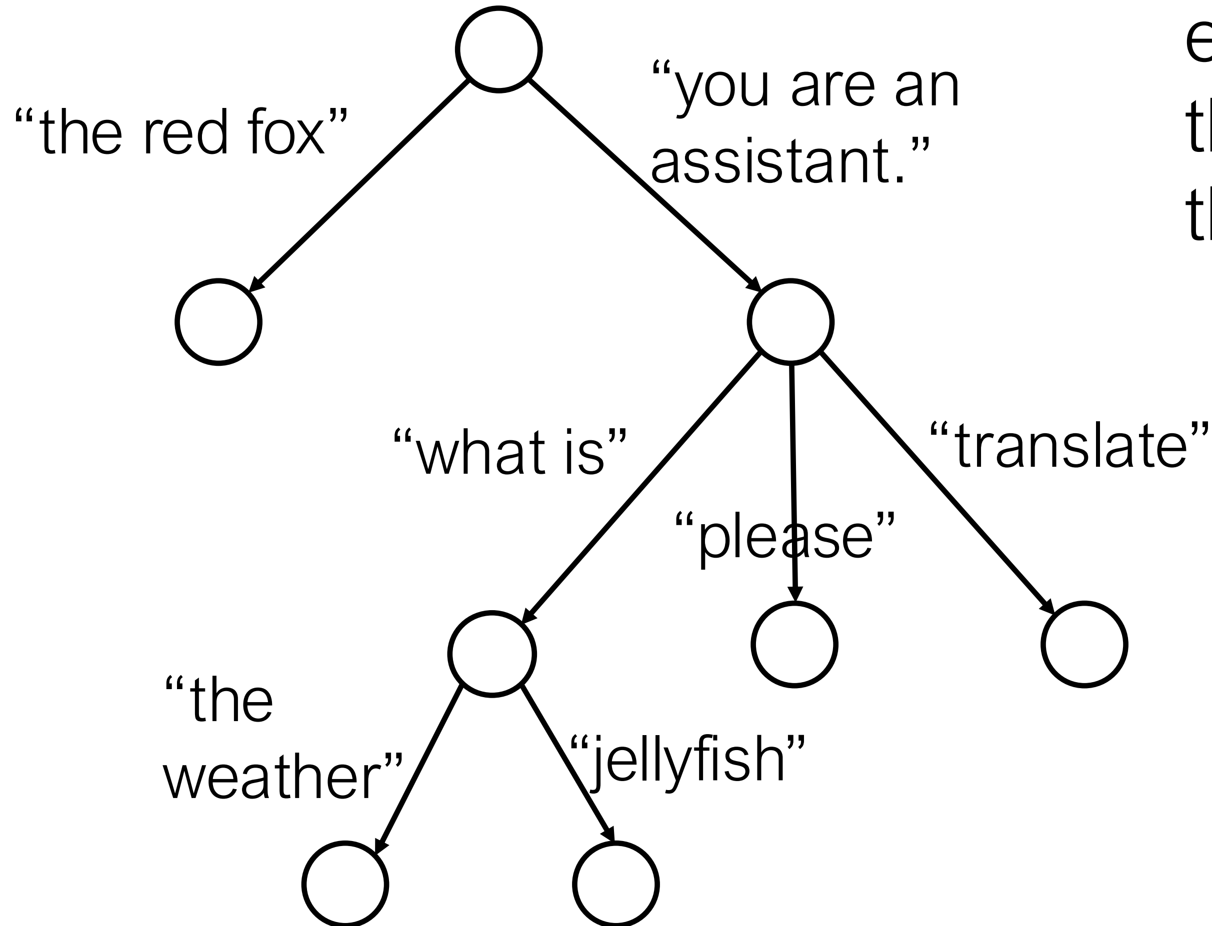
KV Cache take large memory!

- LLaMA3 70B
  - 80 Transformer layers
  - dimension: 8192
  - per token KV uses 2.5MB (stored in FP32)
  - context length 8k => 20GB per user request

Managing KV Cache with RadixAttention

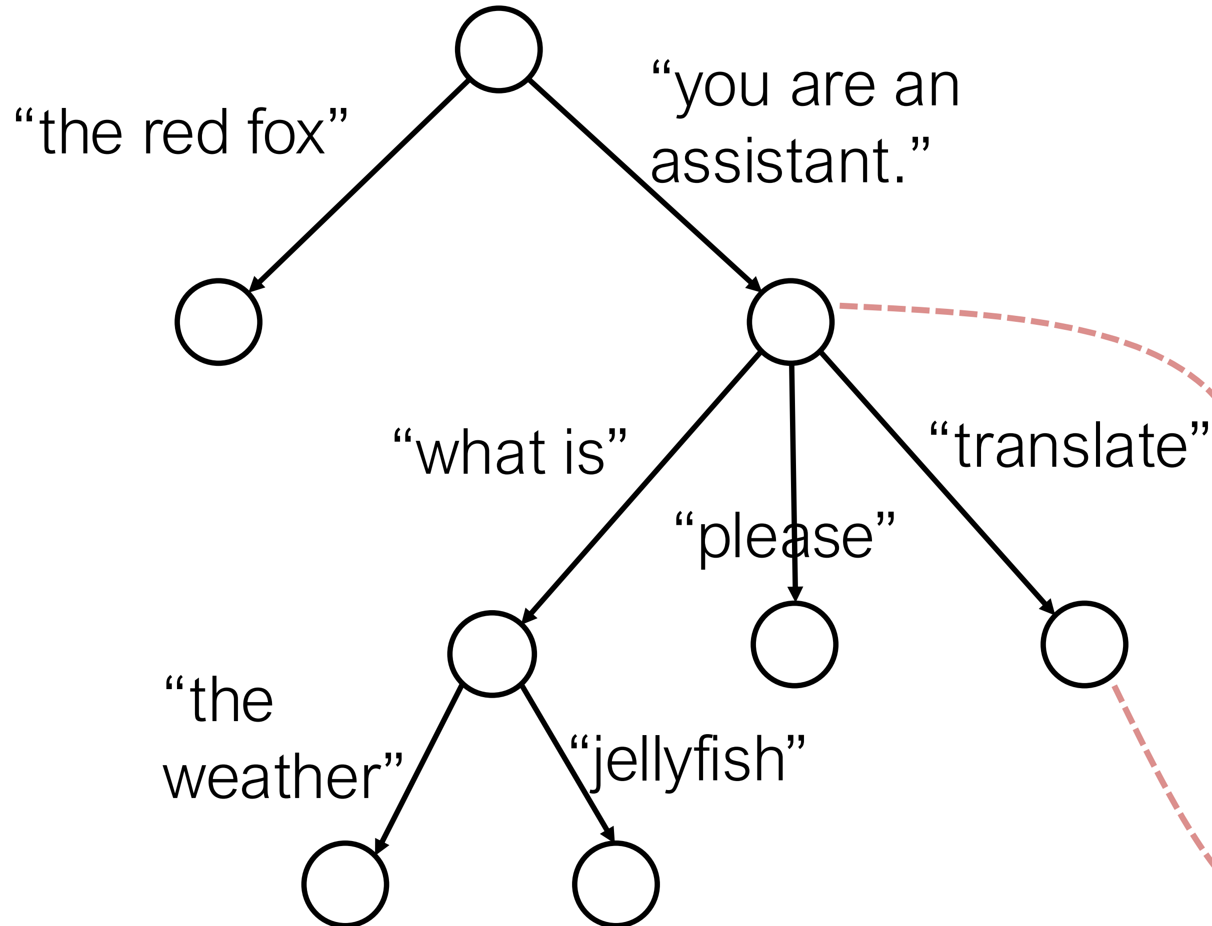
- KV mem pointers are stored in a radix tree (i.e. prefix tree)
  - each edge is a string
  - each node contains memory pointers of KVs
  - the whole path's string represent the prefix
- Retrieve KV cache for a prompt string
  - lookup a prompt string's longest prefix from the radix tree: very fast by matching the edge
  - returns: number of tokens in prefix matched, the matched node

Radix Tree



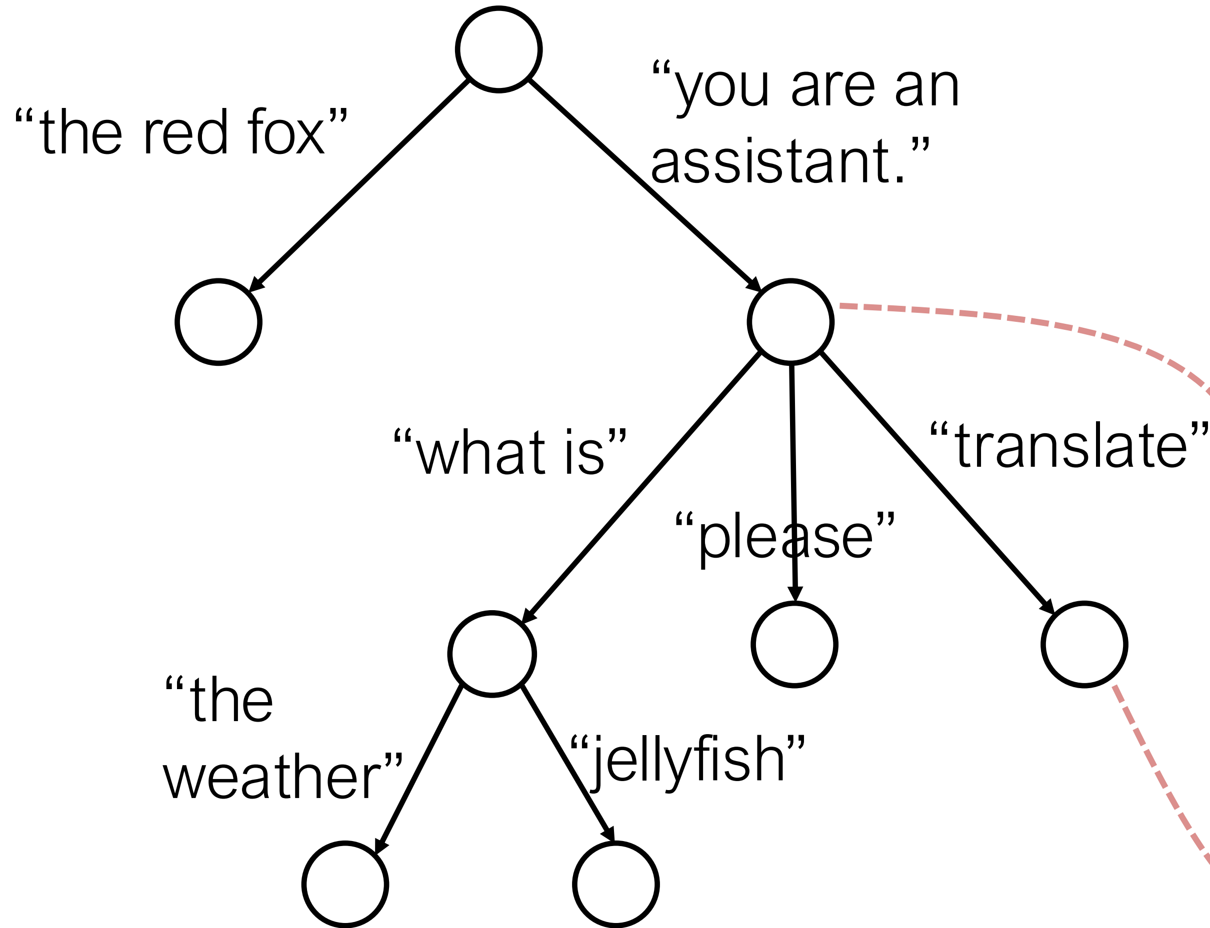
each edge is a string
the whole path’s string represent the prefix

Radix Tree for KV Cache Management

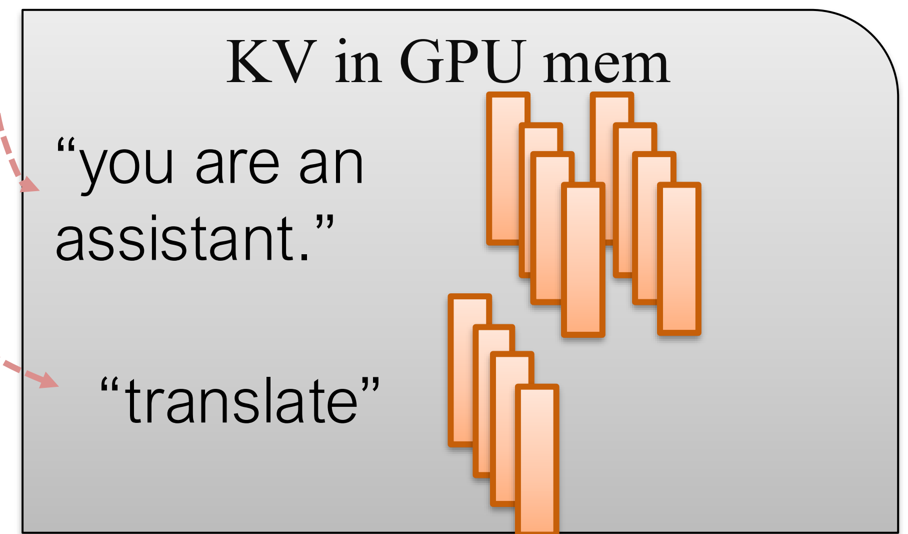


RadixTree is maintained in CPU
each edge is a string
the whole path's string represent the prefix
node points to the GPU memory storing the KVs on the edge

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

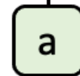


Add KV for New Request Radix Tree

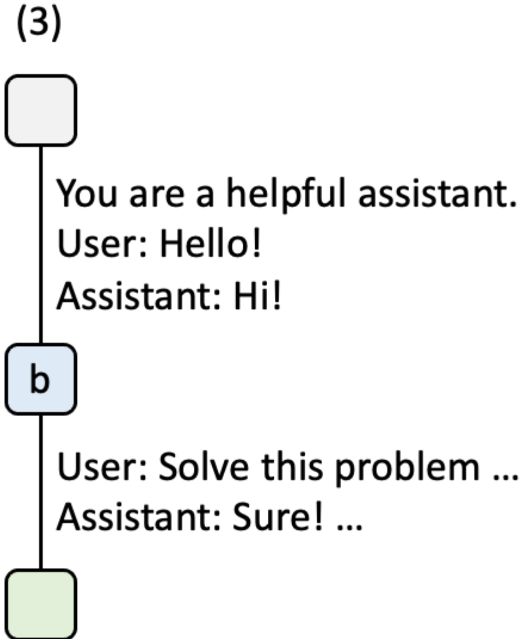
(1)

(2)

You are a helpful assistant.
User: Hello!
Assistant: Hi!

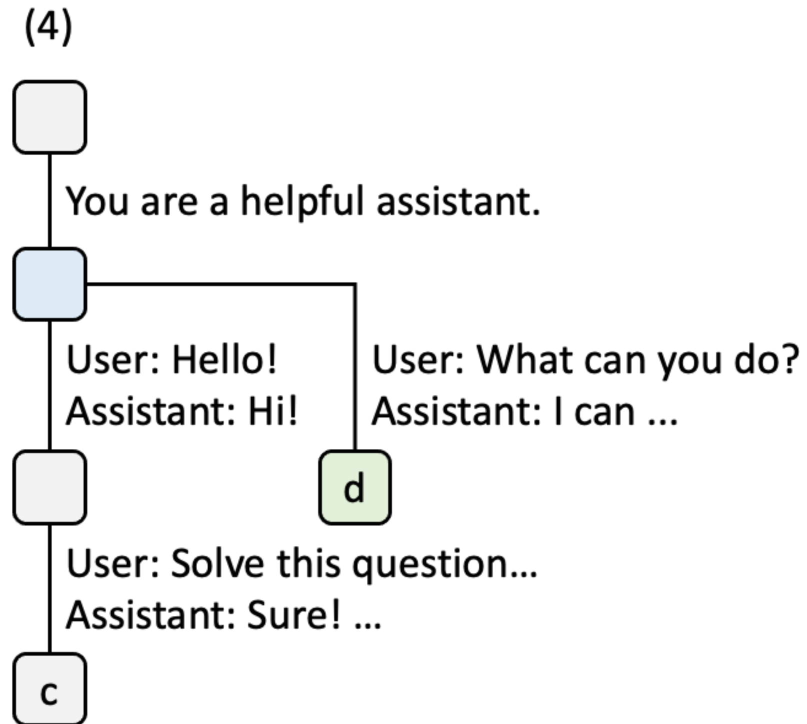


Add KVs for Next Round User Input in Chat

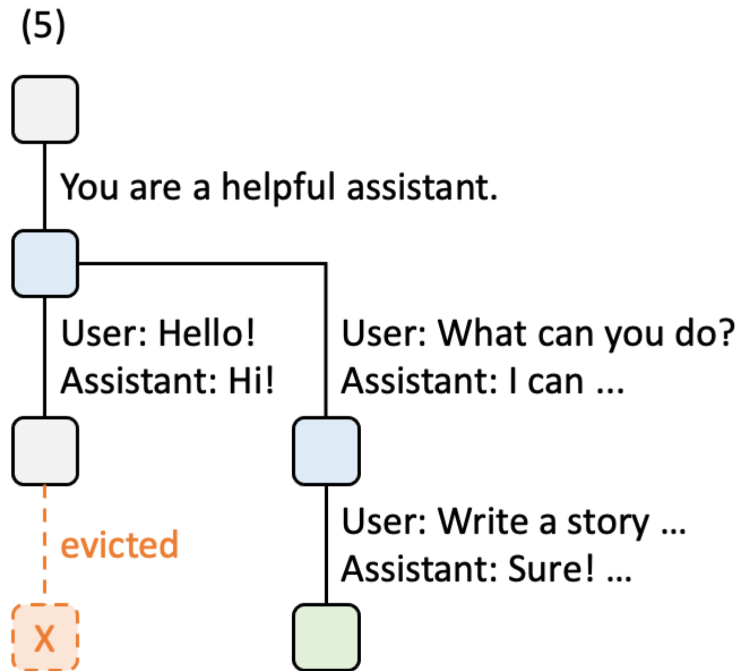


Node Split and Add for a new user's request

node split

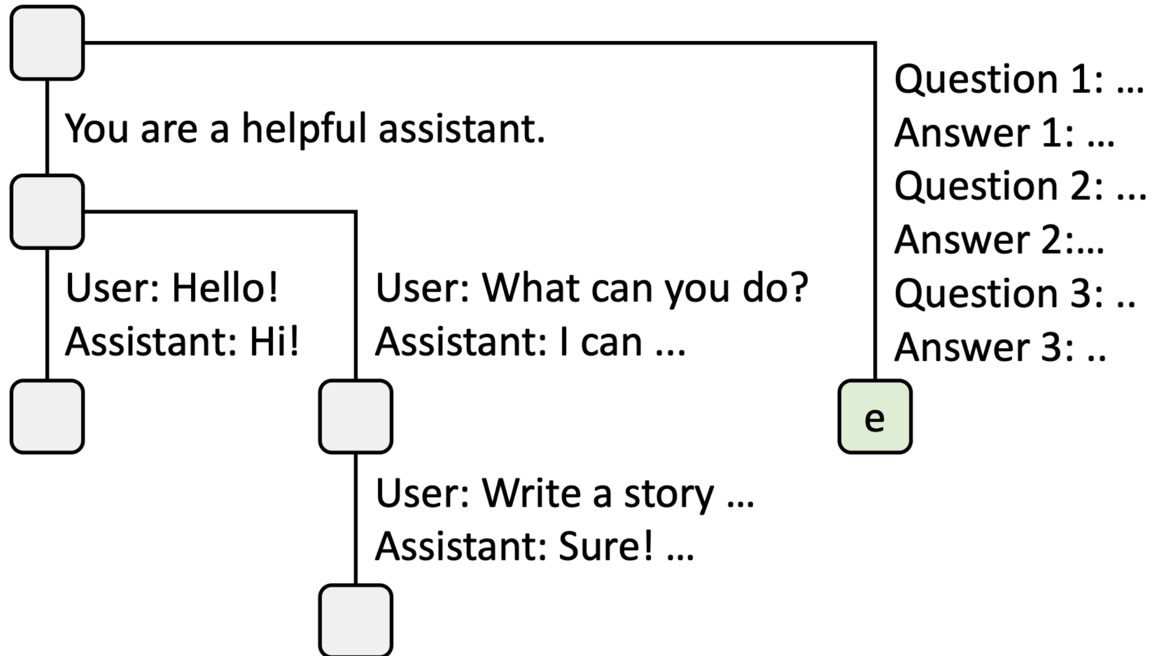


Node Eviction after next round of input

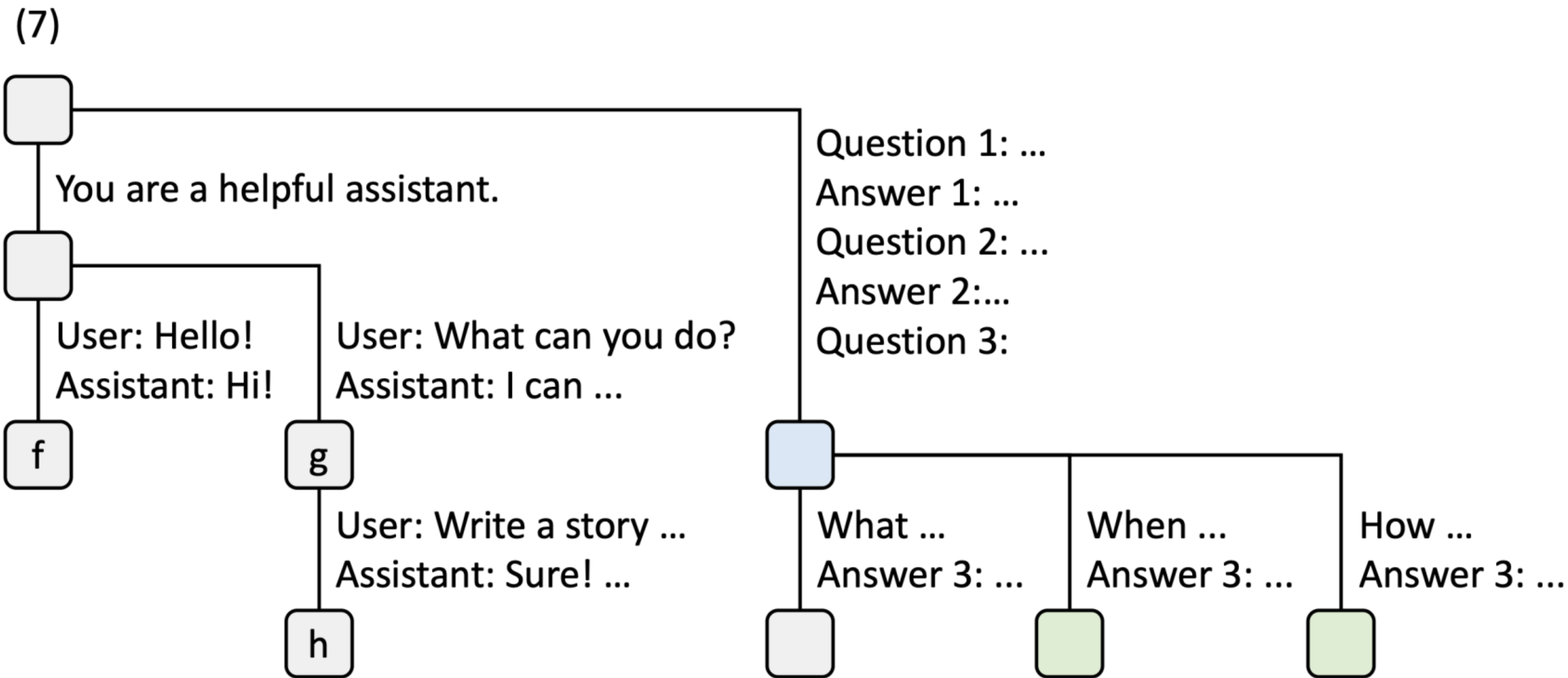


least used KV nodes are evicted from GPU Cache if over flow

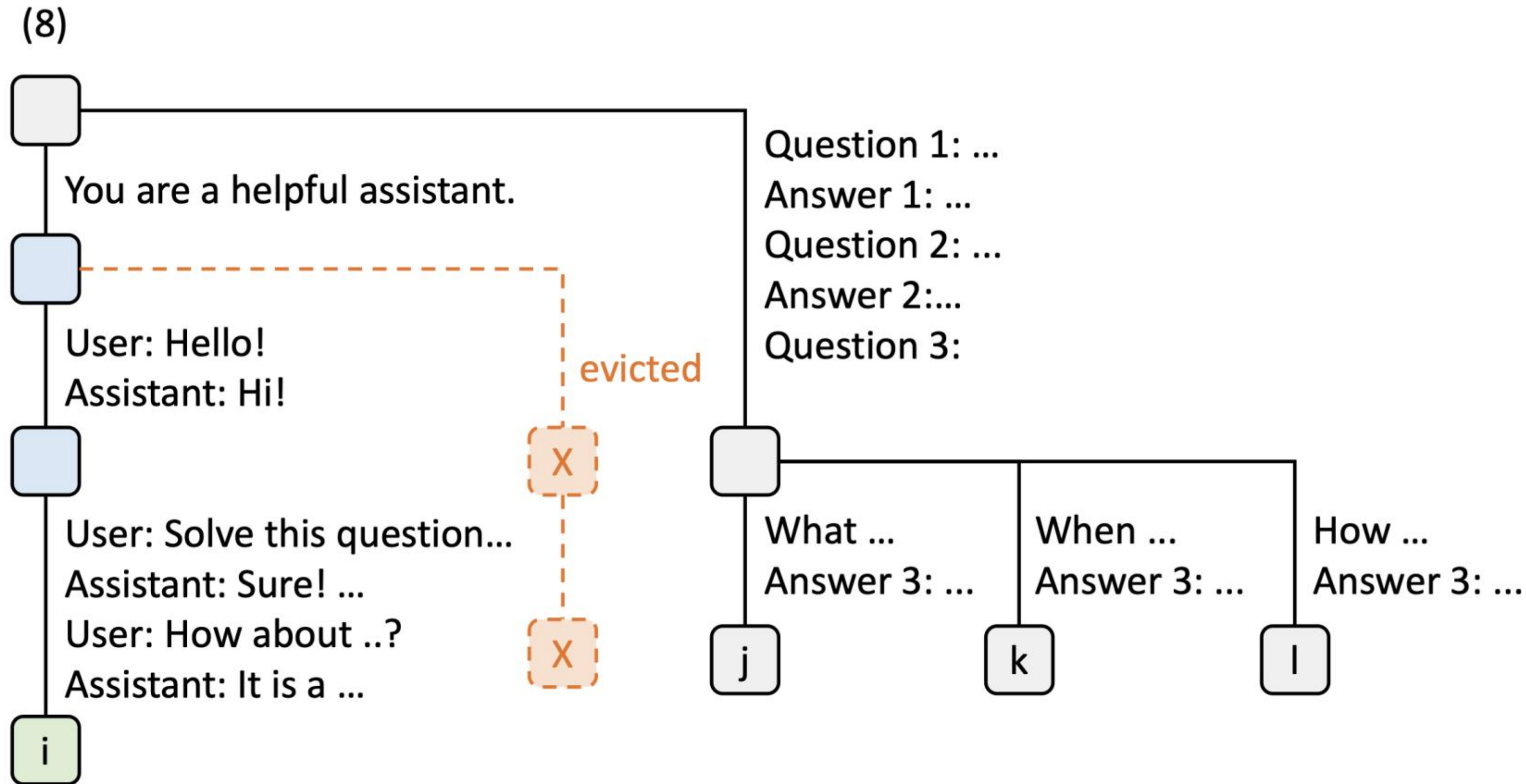
Add KV for User Request with In-context Examples



Add KV for User Request with In-context Examples

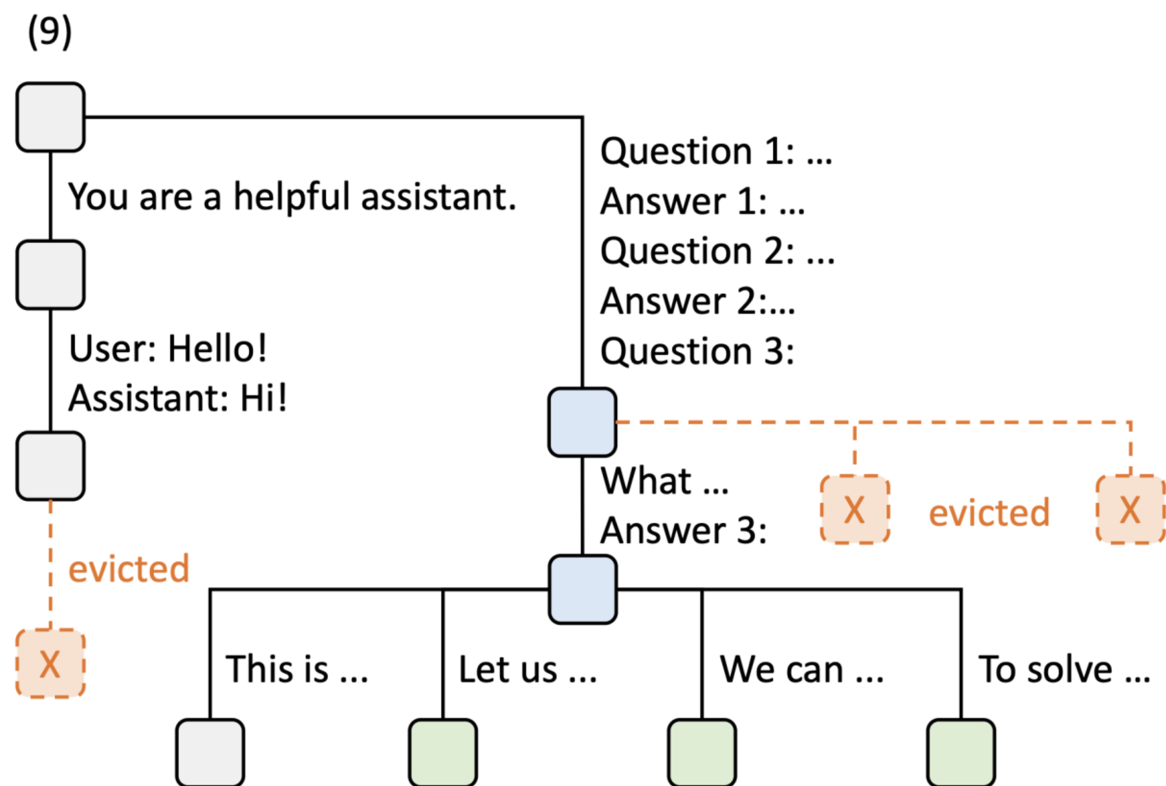


KV Eviction after next round of prior user session



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

Next round of In-context Prompt – Evicting KVs



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

Outline

- Overview of LLM Inference Server
- LLM Request Scheduler
- Scheduler with Continuous Batching
- 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)
```

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: $\frac{\#cached\ tokens}{\#total\ prompt\ tokens}$

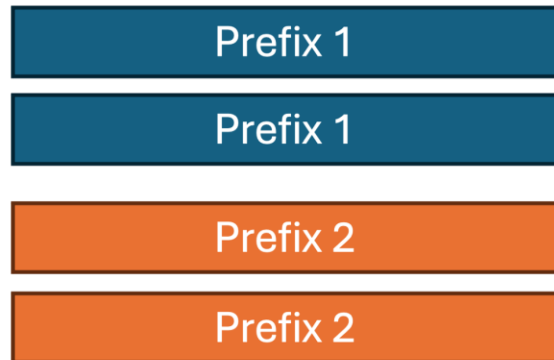
```
def get_next_batch():
    # Match prefix
    for req in waiting_queue:
        req.prefix_length  = match_prefil(req, radix_tree_cache)

    # 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
```

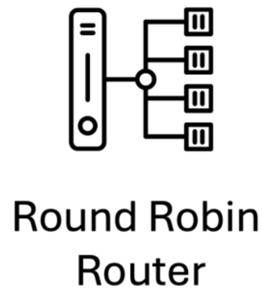
Cache-aware Load Balancer

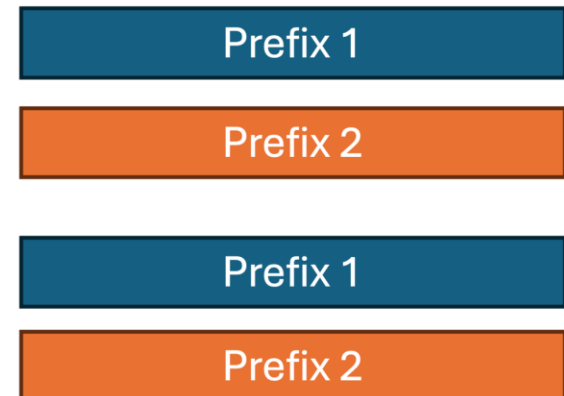
- Maintain KV-Caches on multiple worker nodes

Round Robin Data Parallel (SGLang v0.3)

👎 Not cache aware, ~20% Cache Hit Rate

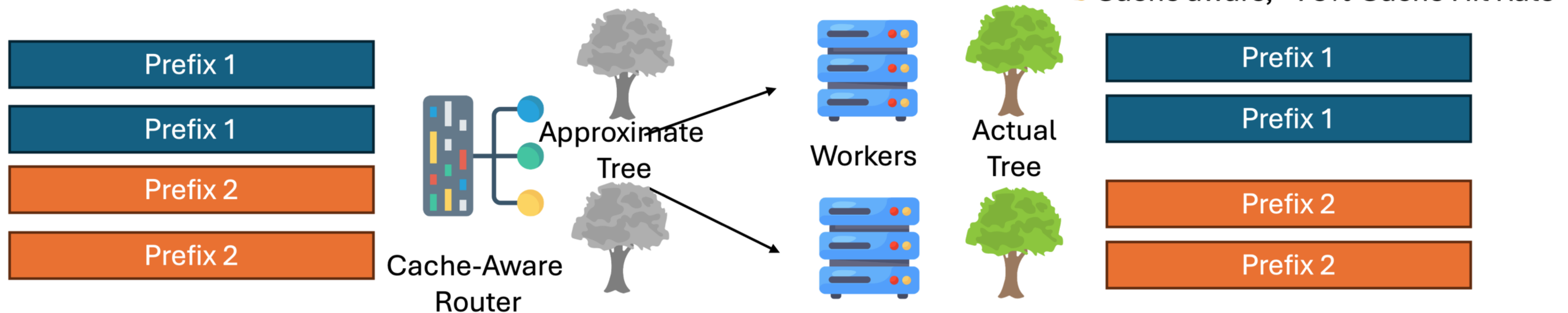




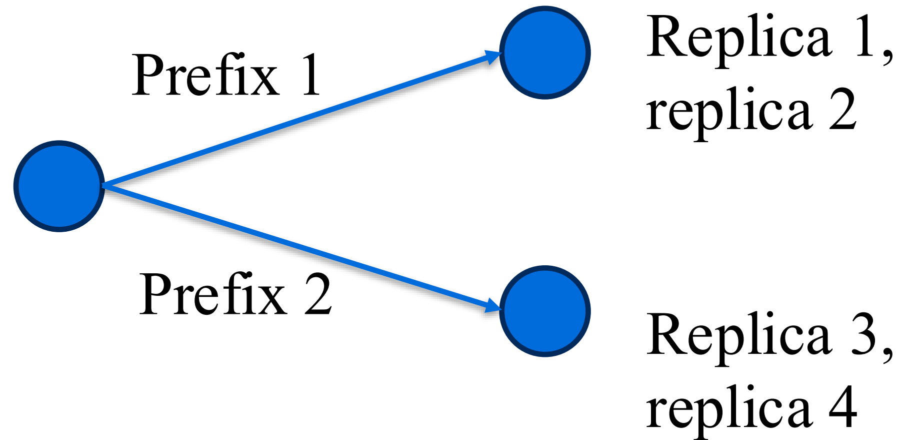


Cache-aware Load Balancer

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



Cache-aware Load Balancer

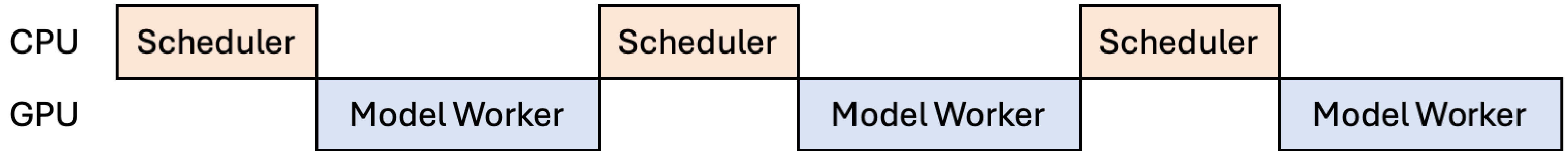


	Round robin	Cache aware load balancer
Throughput (token/s)	82665	158596
Cache hit rate	20%	75%

Outline

- Overview of LLM Inference Server
- LLM Request Scheduler
- Scheduler with Continuous Batching
- 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

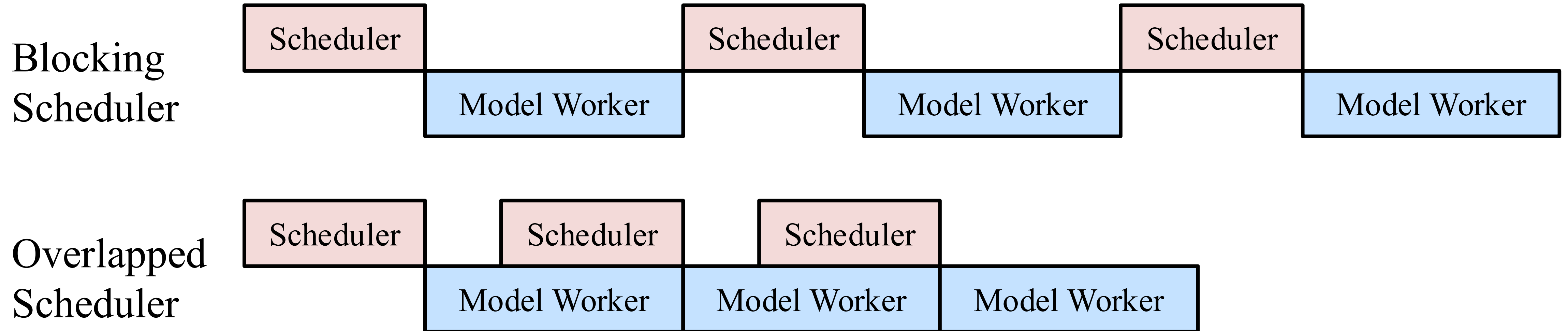
Jobs of the CPU scheduler

- 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

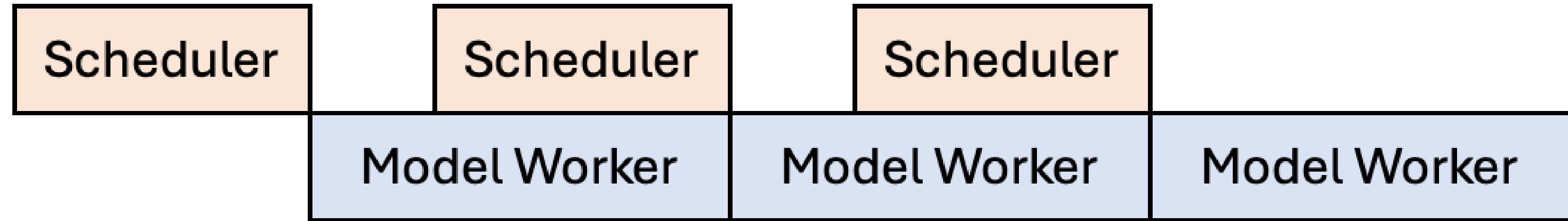
Ideas

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

NanoFlow: Towards Optimal Large Language Model Serving Throughput. Kan Zhu et al.

Zero-overhead CPU scheduling

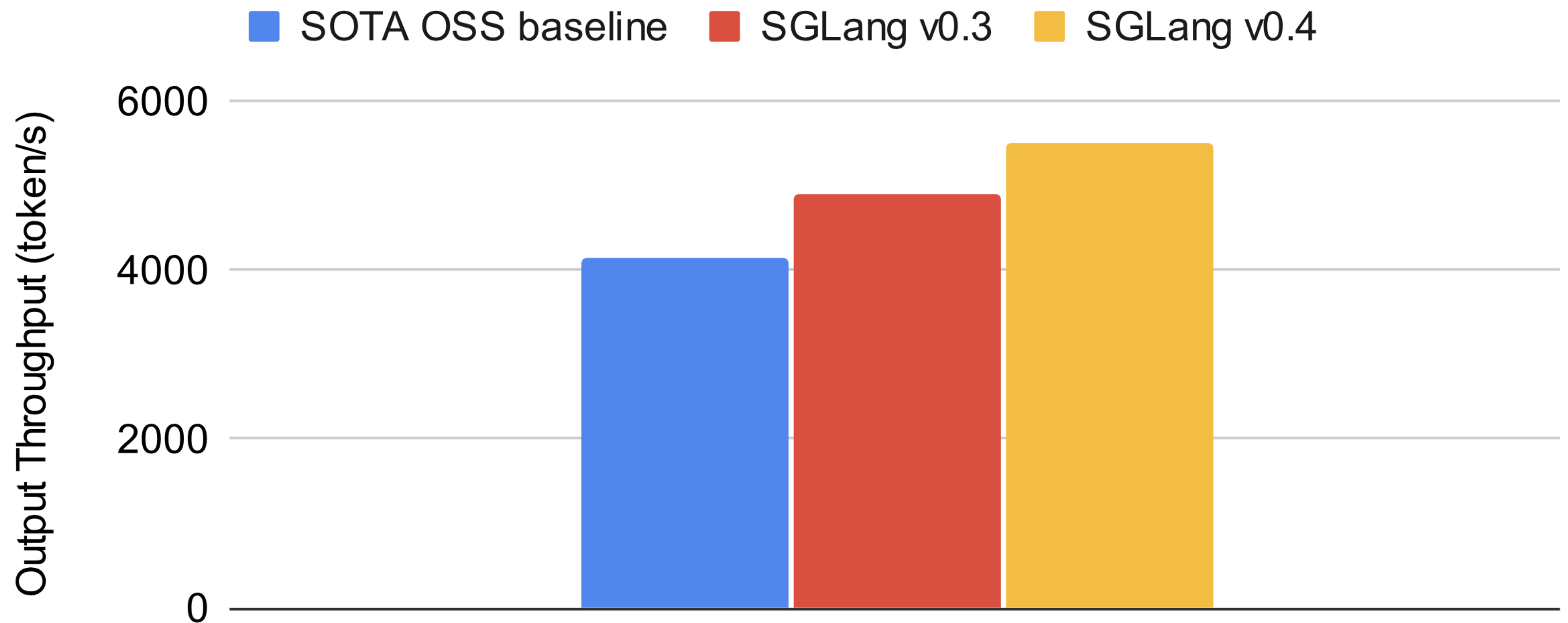
Overlapped Scheduler



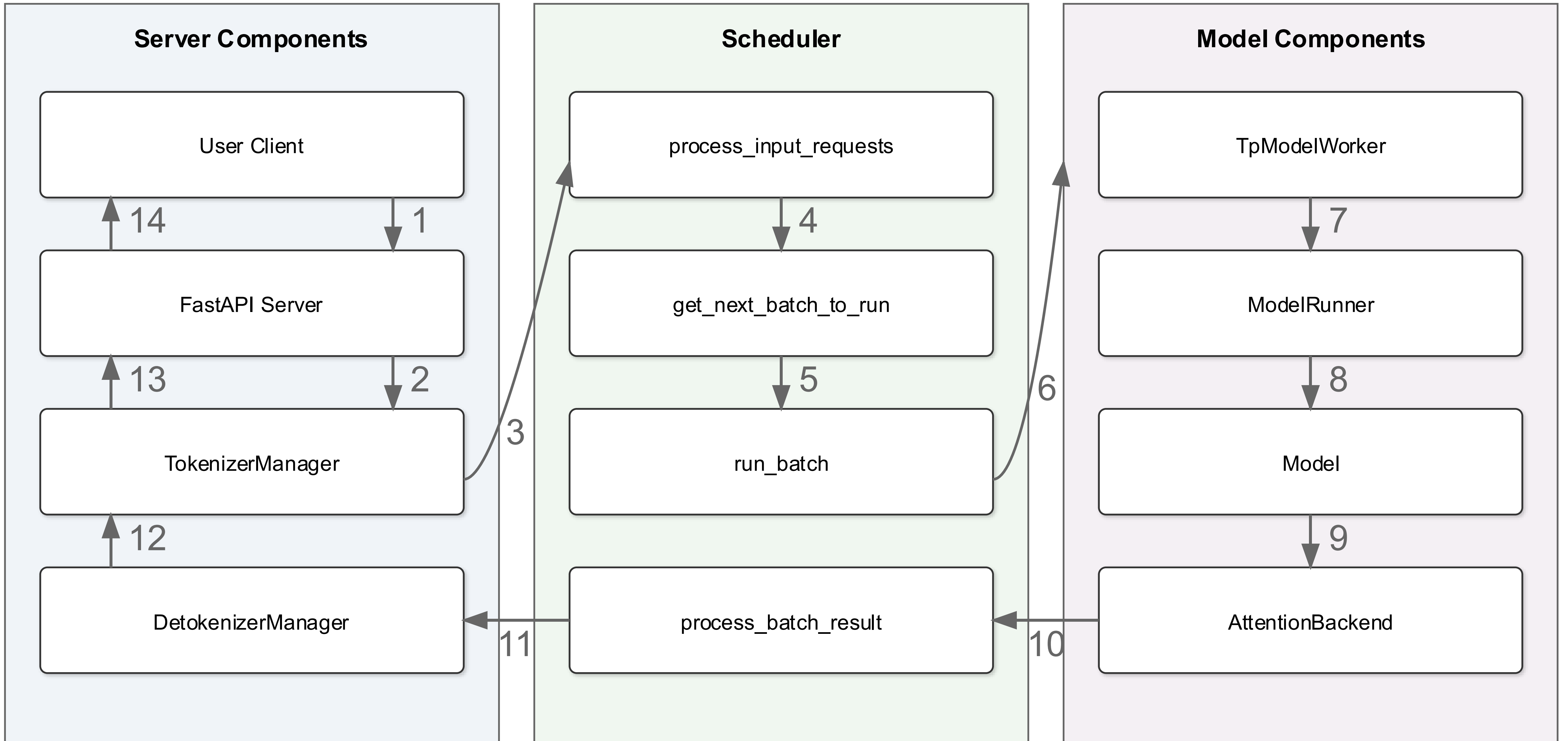
4s
+250ms +255ms +260ms +265ms +270ms +275ms +280ms +2
Processes (106)
[123335] python ...to 100%
CUDA HW (0000:04:00.0 - N Kernel Memory
[All Streams]
99.8% Kernels
0.2% Memory
Threads (15)
[123335] pt_main_thread ...to 100%
OS runtime libraries
Profiler overhead
[124390] pt_main_thread ...to 100%
OS runtime libraries sem_wait sem_wait sem_wait sem_wait sem_wait
CUDA API

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 RadixAttention
   - using radix tree to maintain KVs based on prompt prefix

4. Cache-aware scheduling and load balancing

5. Reducing overhead by overlap scheduling and worker compute

Quiz

- on canvas