Efficient LLM Inference with SGLang

Speaker:

Ying Sheng (xAI, LMSYS, UCLA)

Content

SGLang overview

Techniques covered in this talk

- Efficient KV cache reuse with RadixAttention
- Cache-aware load balancing with SGL Router
- Zero-overhead CPU scheduling
- Hierarchical KV cache

Open-source community and industry deployment

- Hot topics in LLM inference systems and roadmap
- Large scale deployment practice – A case study of DeepSeek inference system

SGLang Overview

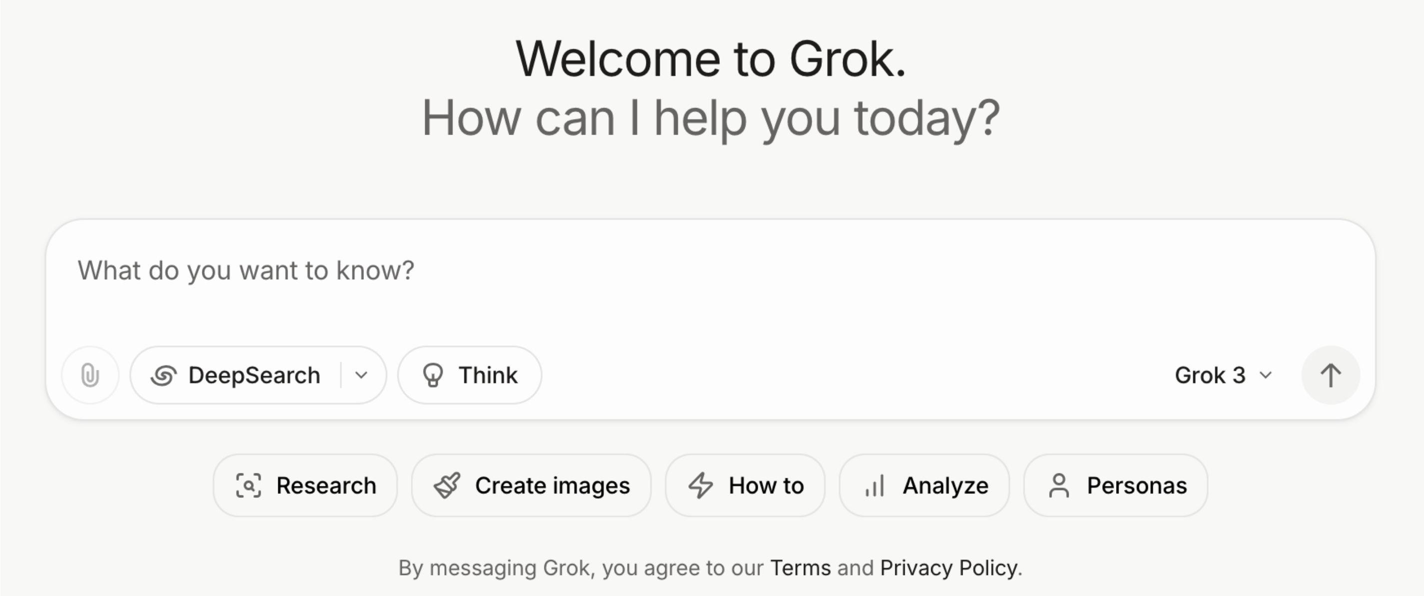
SGLang is a fast serving framework for large language models and vision language models.

What is SGLang?

An open-source **inference engine** for LLMs 

Comes with its **unique features** for better performance

Serves the **production and research** workloads at xAI (100K+ GPUs)



https://grok.com/

SGLang provides leading inference performance

Compared to the other popular inference engines:

v0.1 (Jan. 2024)

5x higher throughput with automatic KV cache reuse
3x faster grammar-based constrained decoding

v0.2 (July 2024)

3x higher throughput with low-overhead CPU runtime

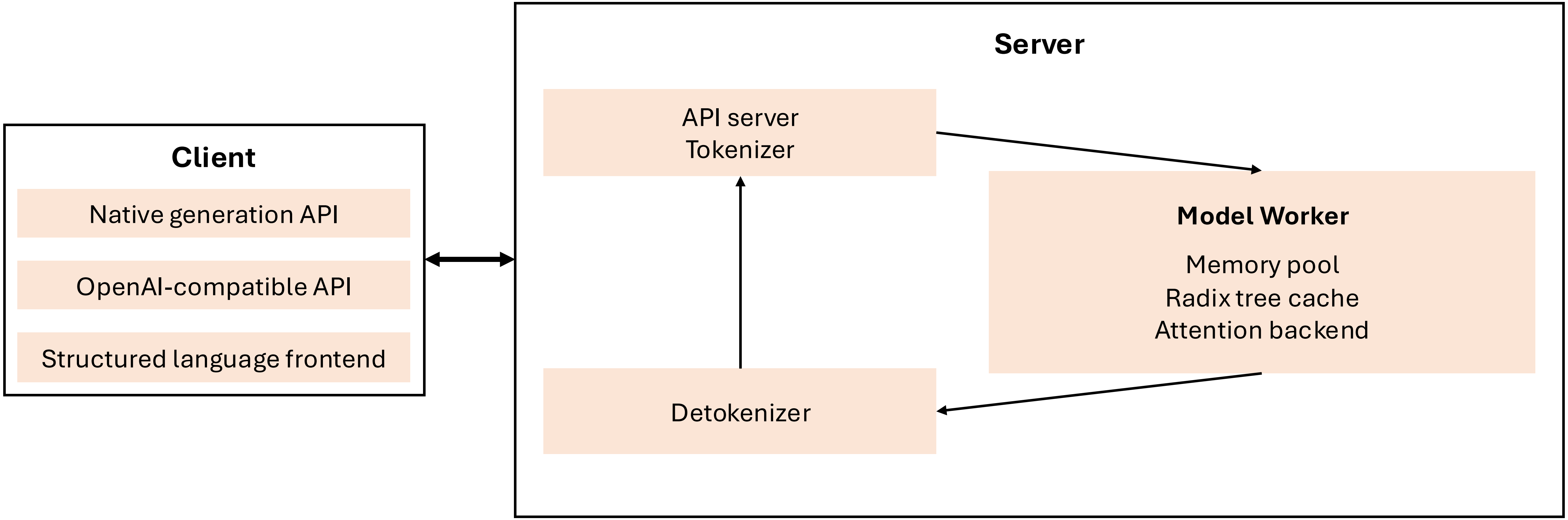
v0.3 (Sept. 2024)

7x faster triton attention backend for custom attention variants (DeepSeek MLA)
1.5x lower latency with torch.compile

v0.4 (Dec. 2024)

Zero-overhead CPU scheduler and structured outputs
Cache-aware load balancer
Fastest DeepSeek inference

SGLang architecture overview



Lightweight and customizable codebase in Python/PyTorch

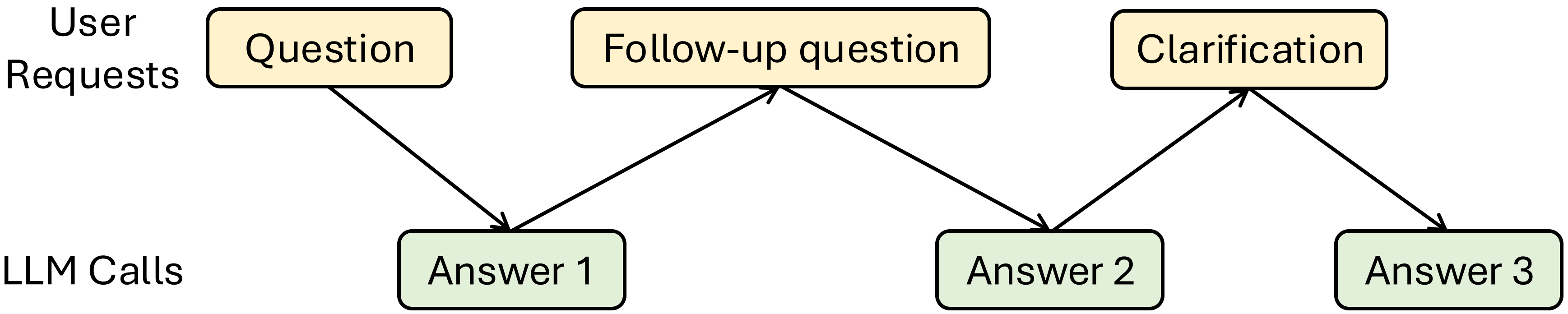
Major Techniques

Four techniques covered in this talk

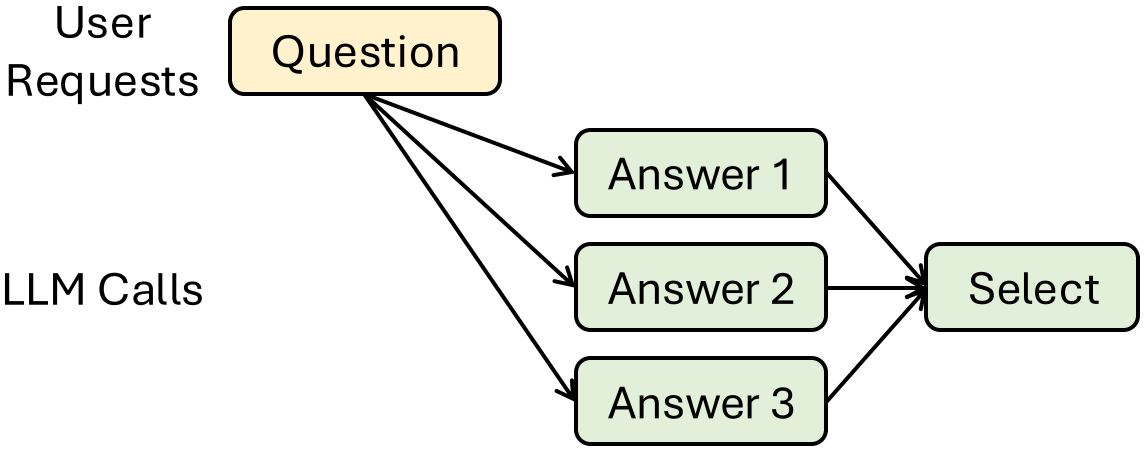
1. Efficient KV cache reuse with RadixAttention
2. Cache-aware load balancing with SGL-Router
3. Zero-overhead CPU scheduling
4. Hierarchical KV cache

LLM inference pattern:
Complex pipeline with multiple LLM calls

Chained calls



Parallel calls



LLM inference pattern:
Complex pipeline with multiple LLM calls

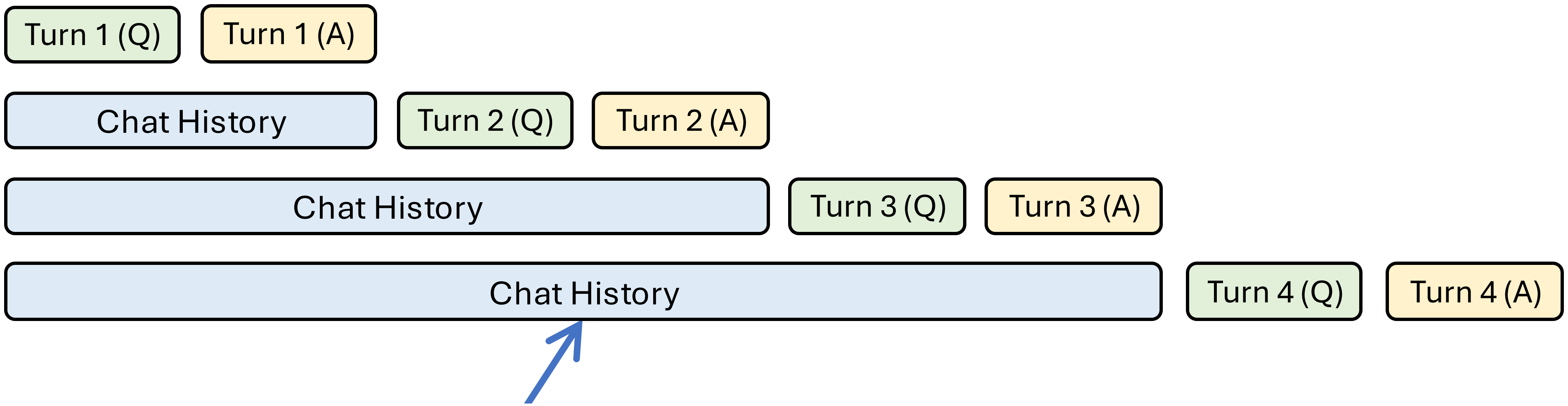
Chained calls

Parallel calls

Multi-call structure **brings optimization opportunities**
(e.g., caching, parallelism, shortcut)

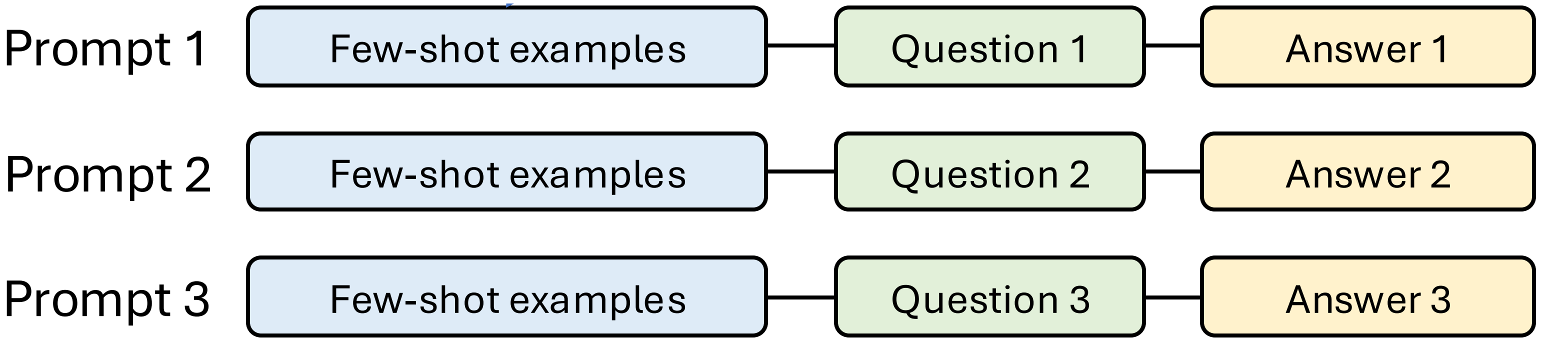
There are rich structures in LLM calls

(a) Multi-turn chat



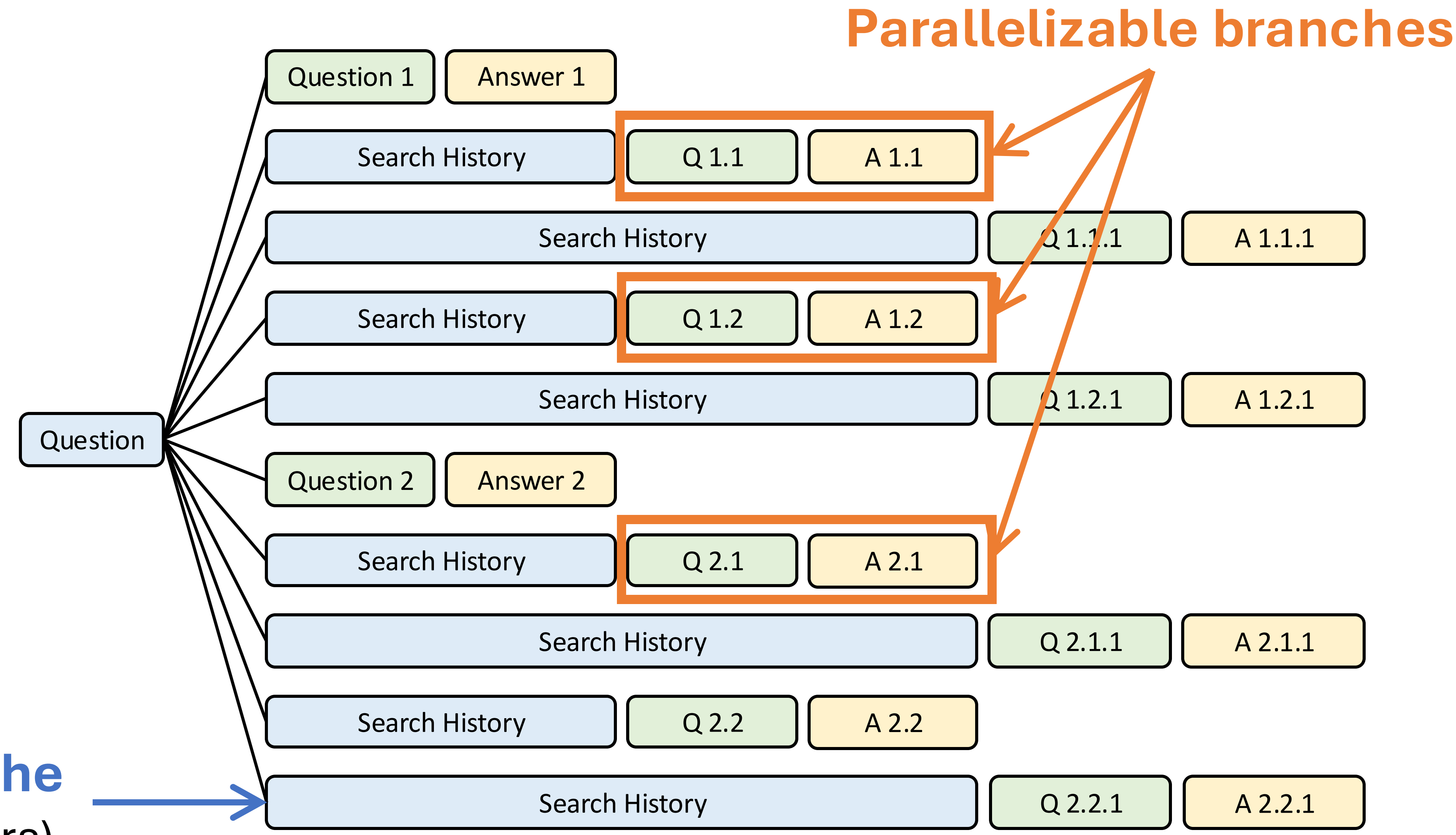
Reusable KV cache (Key-Value cache, some intermediate tensors)

(b) Few-shot learning



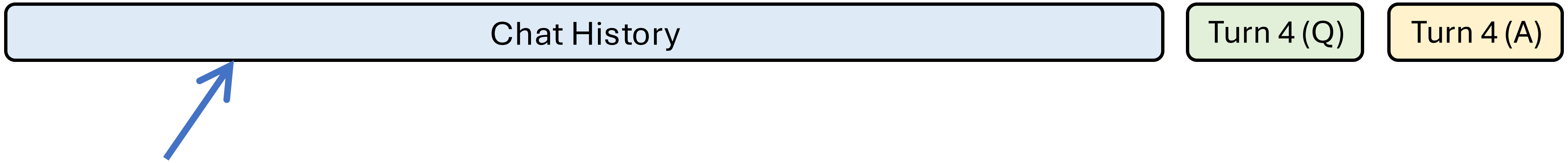
The structures can be very complicated

(c) Tree search with LLM agents



Reusable KV cache
(intermediate tensors)

Technique 1: Efficient KV cache reuse with RadixAttention



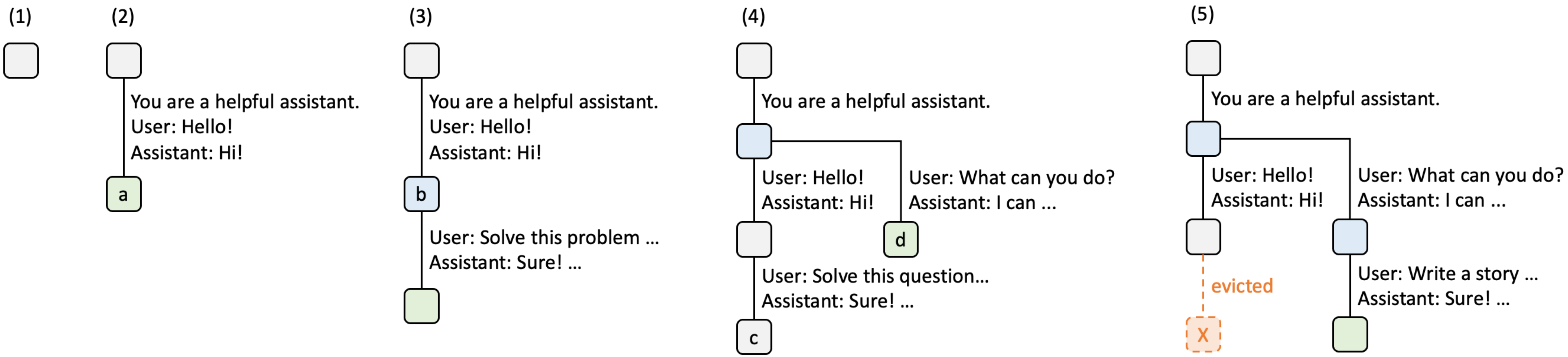
KV Cache

- Some reusable intermediate tensors
- Can be very large (>20GB, larger than model weights)
- Only depends on the prefix tokens

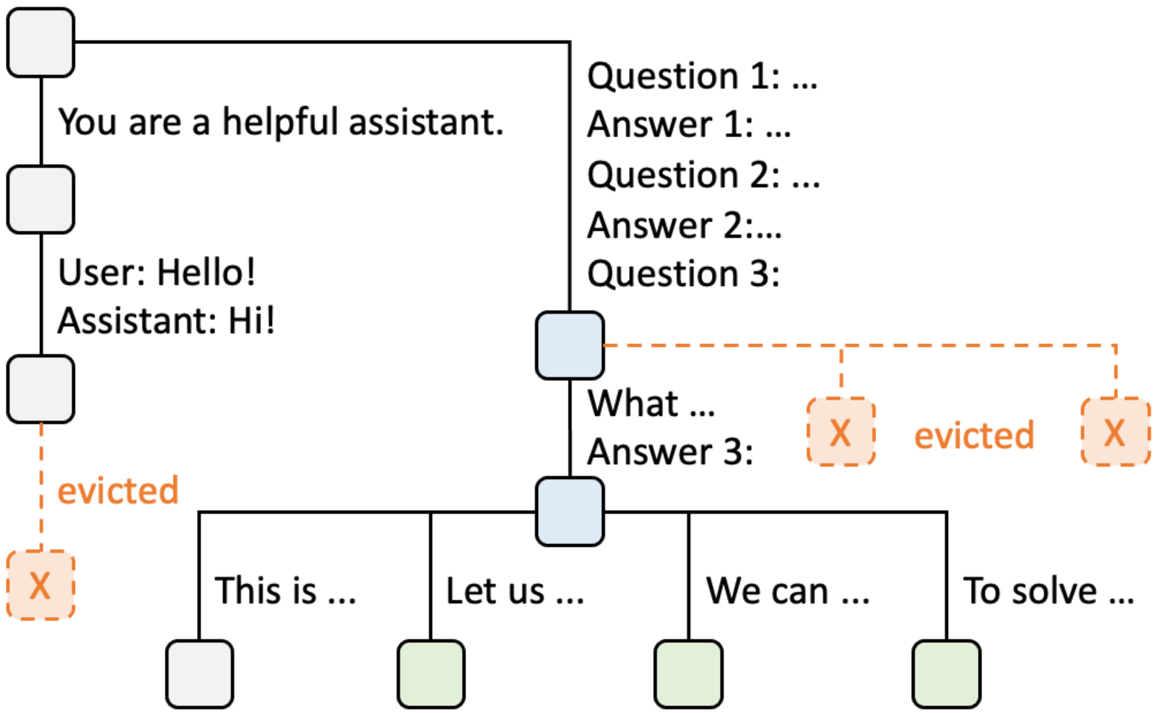
Existing systems: Discard KV cache after an LLM call finishes

Ours: Maintain the KV cache of all LLM calls in a radix tree (compact prefix tree)

RadixAttention maintains the KV cache of all LLM calls in a radix tree (compact prefix tree)



RadixAttention handles complex reuse patterns



RadixAttention enables **efficient prefix matching, insertion, and eviction.**

It handles trees with hundreds of thousands of tokens.

Cache-aware scheduling increases cache hit rate

Idea: Utilize **user annotations** and **runtime metrics** for scheduling

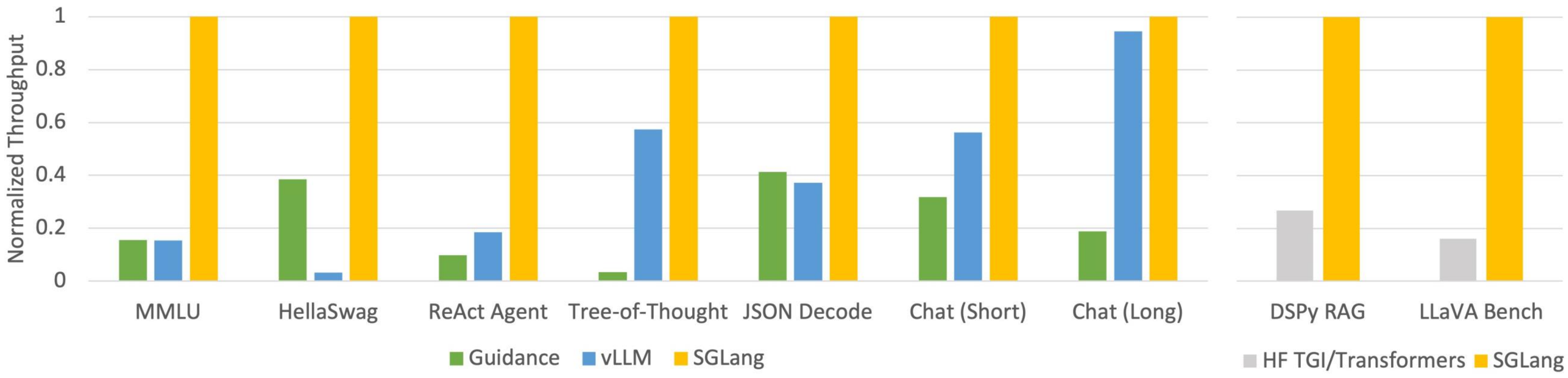
Single worker case

Sort the requests in the queue according to matched prefix length

```
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
```

Results

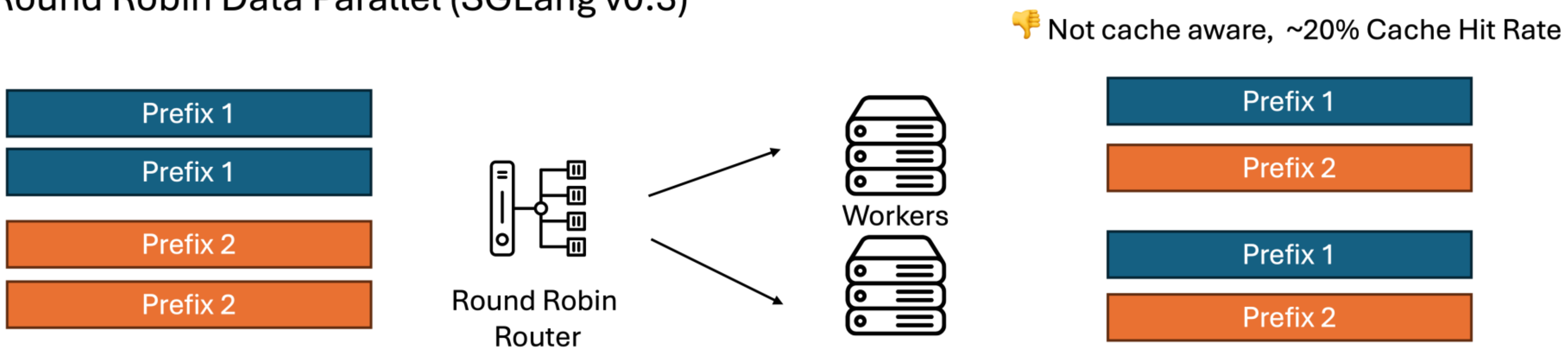
- Up to **5x higher throughput** with KV cache reuse and parallelism
- Works automatically across workloads and text/image tokens



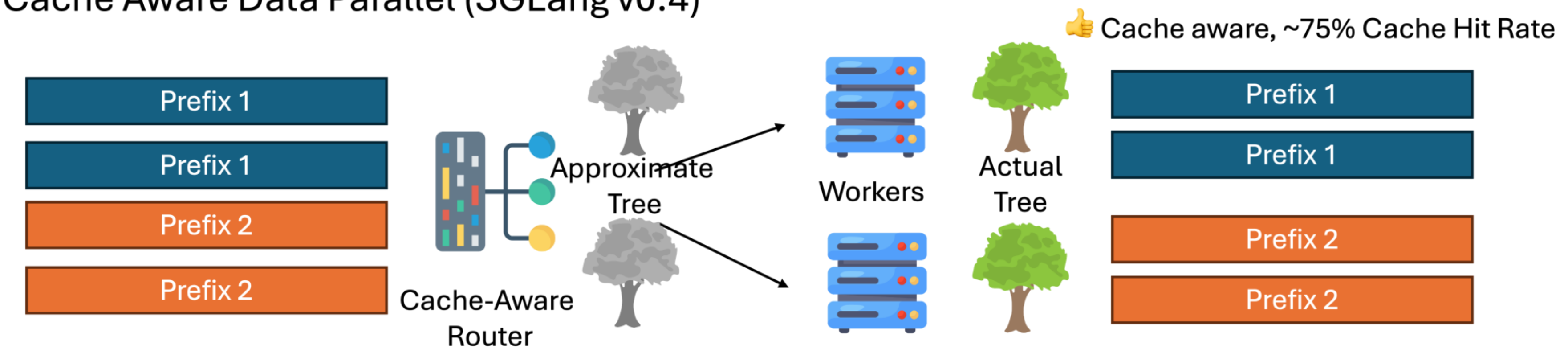
Source: SGLang v0.1 blog, https://lmsys.org/blog/2024-01-17-sglang/

Technique 2: Cache-aware load balancer - SGL Router

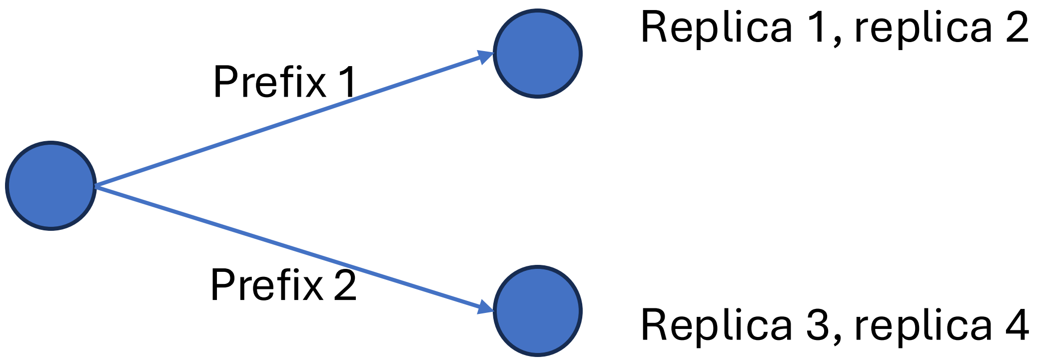
Round Robin Data Parallel (SGLang v0.3)



Cache Aware Data Parallel (SGLang v0.4)



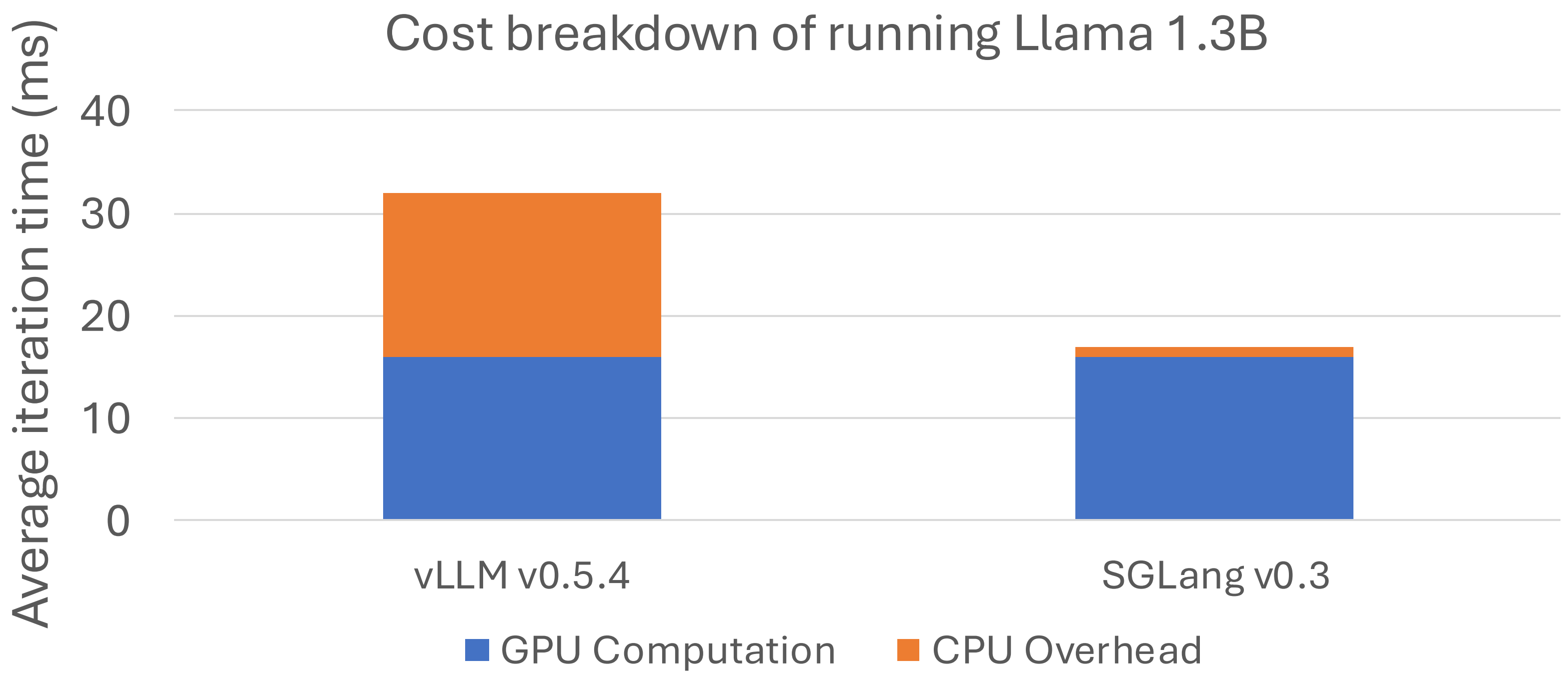
Technique 2: Cache-aware load balancer - SGL Router



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

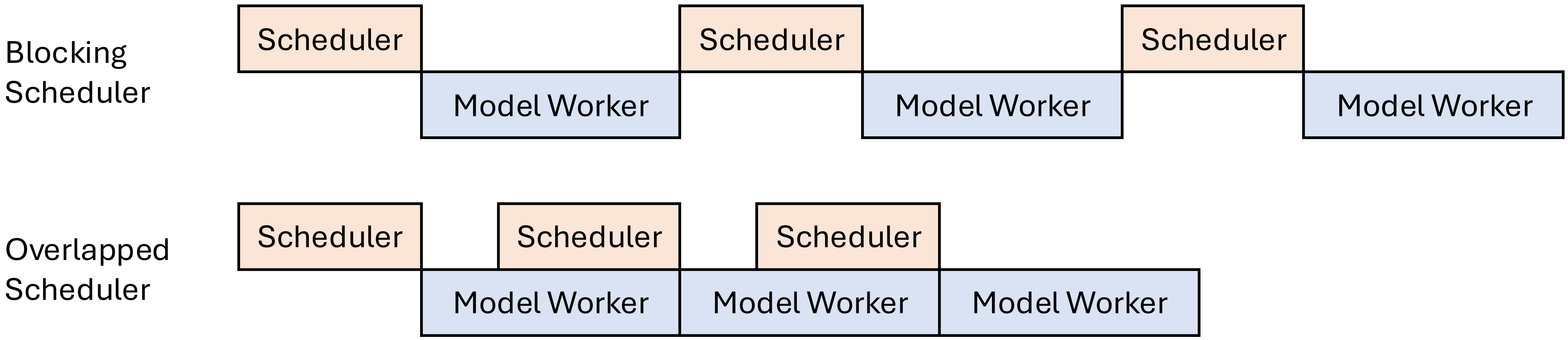
Technique 3: Zero-overhead CPU scheduling

An unoptimized inference engine can waste more than 50% time on CPU scheduling.



Source: https://mlsys.wuklab.io/posts/scheduling_overhead/

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

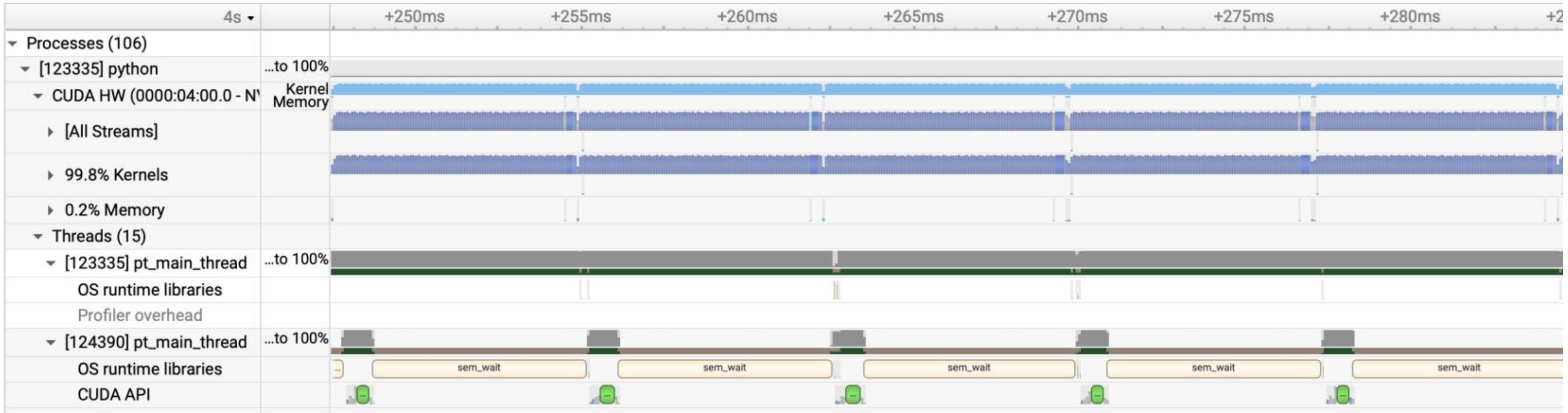
- Resolve the dependency by delaying the stop condition check
- Use CUDA events and streams to do fine-grained scheduling

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

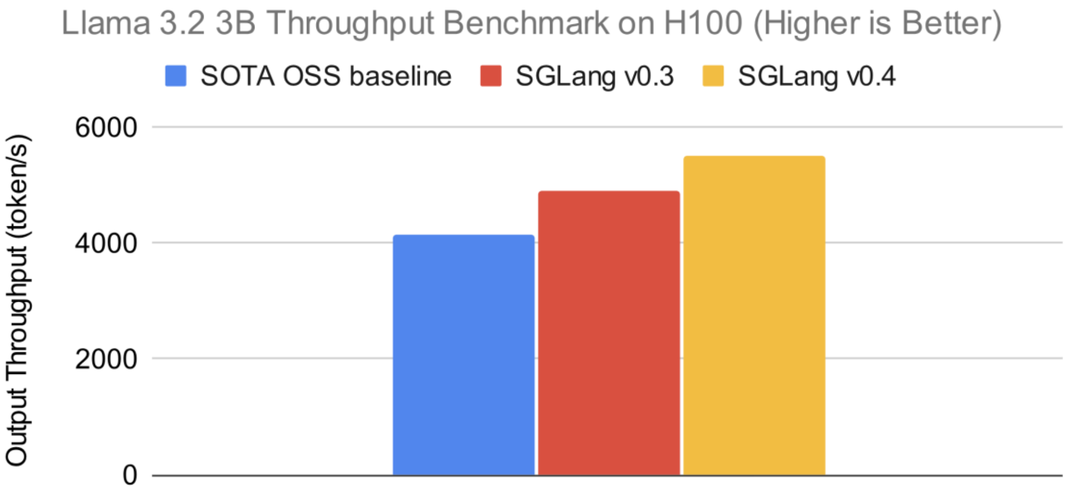
```python
while True:
    recv_reqs = self.recv_requests()
    self.process_input_requests(recv_reqs)

    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
```

Performance

Zero CPU time according to the nsight profiler



1.3x faster than SOTA OSS baseline



Technique 4: Hierarchical KV cache

Skipped

Open-Source Community and Industry Deployment

Industry adoption

SGLang has been deployed to large-scale production, generating **trillions of tokens** every day. It is supported by the following institutions (incomplete list) with **400+** OSS contributors:

Feature coverage

- **Support all common optimizations**
  - Continuous batching, prefix caching, token attention (paged attention), speculative decoding, tensor parallelism, chunked prefill, structured outputs, quantization (FP4/FP8/INT4), multi-lora serving

- **Support all major OSS models**
  - Text: DeepSeek V3/R1, Llama 1/2/3, Qwen, Mixtral
  - Vision: Llama 4, QwenVL, DeepSeek-VL, Llava, Pixtral

- Documentation w/ interactive notebooks: https://docs.sglang.ai/

Open-source development roadmap

Throughput-oriented large-scale deployment

Prefill-decode disaggregation (link)
Expert/pipeline/context parallelism

Reinforcement learning integration

Weight sync, asynchronous algorithms, memory saver
veRL Integration (link), Areal, LlamaFactory

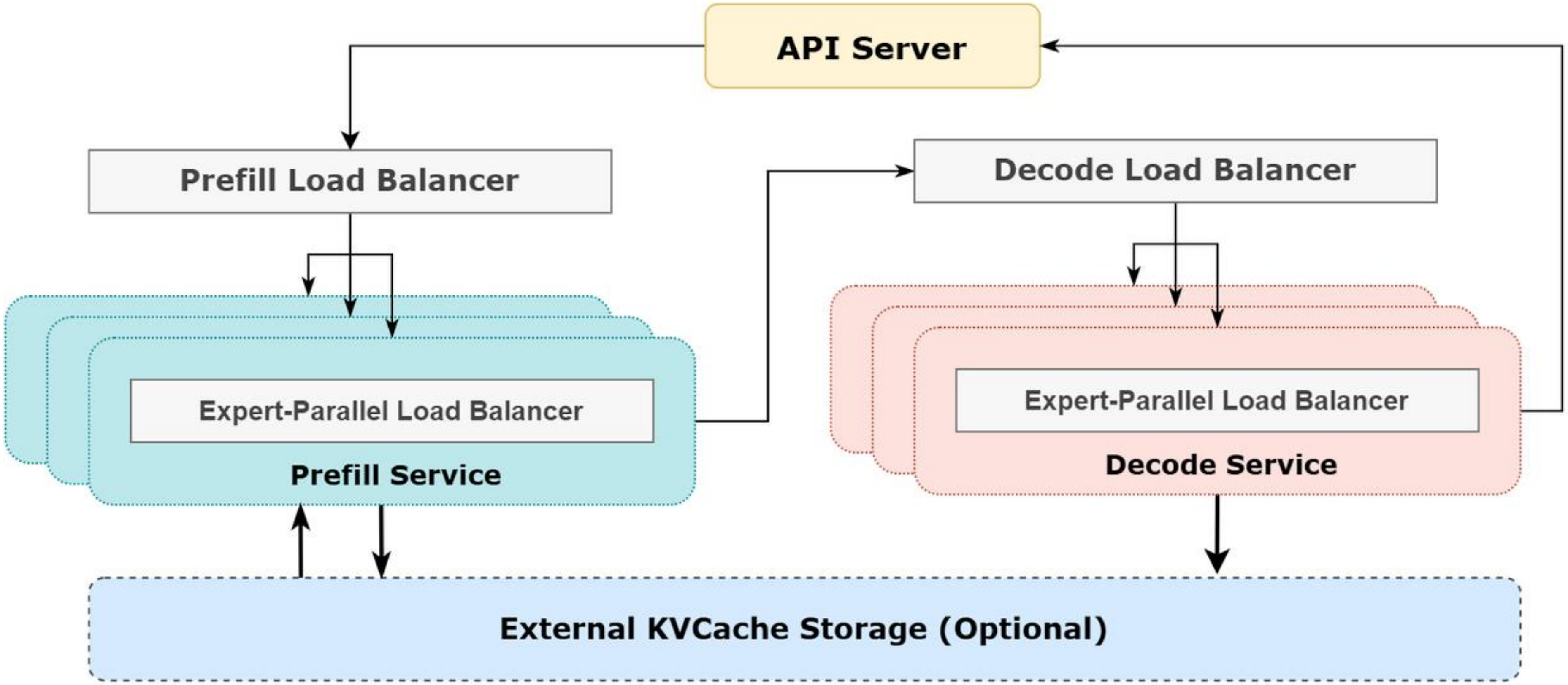
Low latency optimizations

Speculative decoding (e.g., EAGLE 3), kernel optimizations

The full roadmap: https://github.com/sgl-project/sglang/issues/4042

A case study of the DeepSeek system

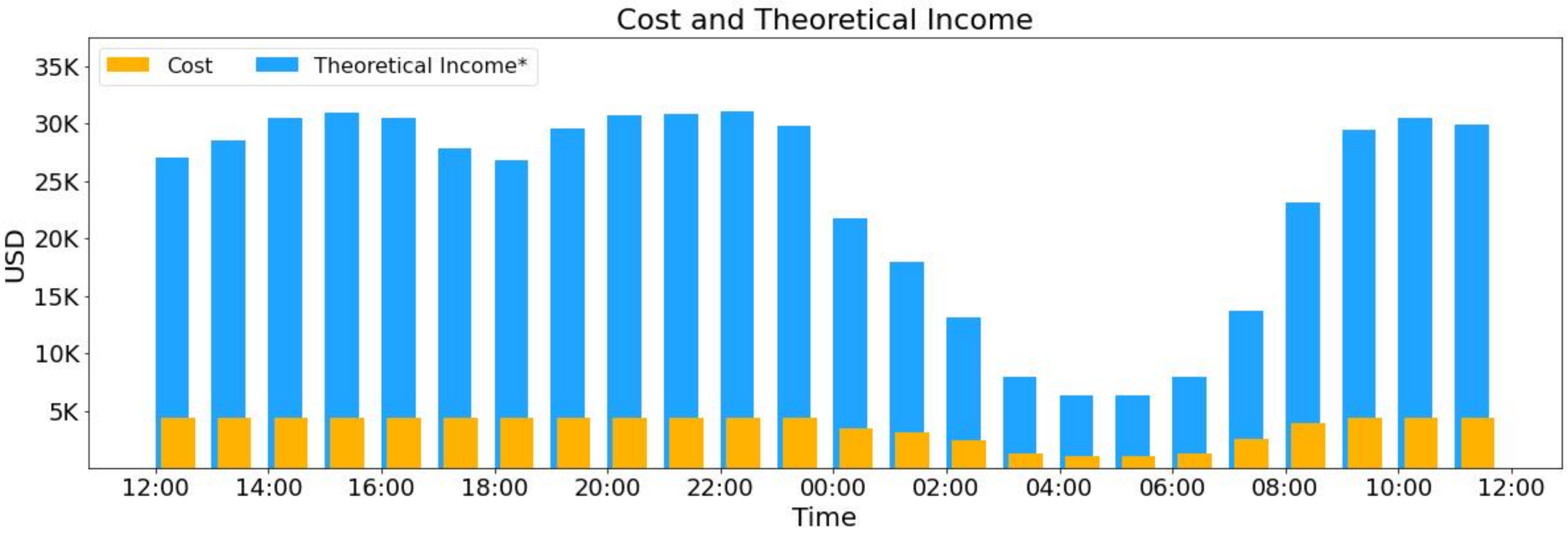
Load balancer + prefill / decode disaggregation + speculative decoding + quantization + tensor/expert/pipeline parallelism + external KVCache storage



A case study of the DeepSeek system

Within the 24-hour statistical period
- Total input tokens: 608B, of which 342B tokens (56.3%) hit the on-disk KV cache.
- Total output tokens: 168B.

If all tokens were billed at DeepSeek-R1's pricing, the total daily revenue would be $562,027, with a cost profit margin of 545%.



* The theoretical income is calculated based on R1's standard API pricing, taking into account all tokens across web, APP, and API. It is not our actual income.

Question & Answer

Github: https://github.com/sgl-project/sglang

X (twitter): https://x.com/lmsysorg

Paper (NeurIPS'24) : https://arxiv.org/abs/2312.07104

Welcome to join the slack and bi-weekly dev meeting!