Efficient LLM Inference with SGLang

Lianmin Zheng

xAI

Content

SGLang overview

Major techniques

- Efficient KV cache reuse with RadixAttention
- Efficient constrained decoding with compressed finite state machine
- Low-overhead CPU scheduling
- Torch native optimizations (torch.compile, torchao)

Preliminary benchmark results on MI300

Open-source community and roadmap

SGLang Overview

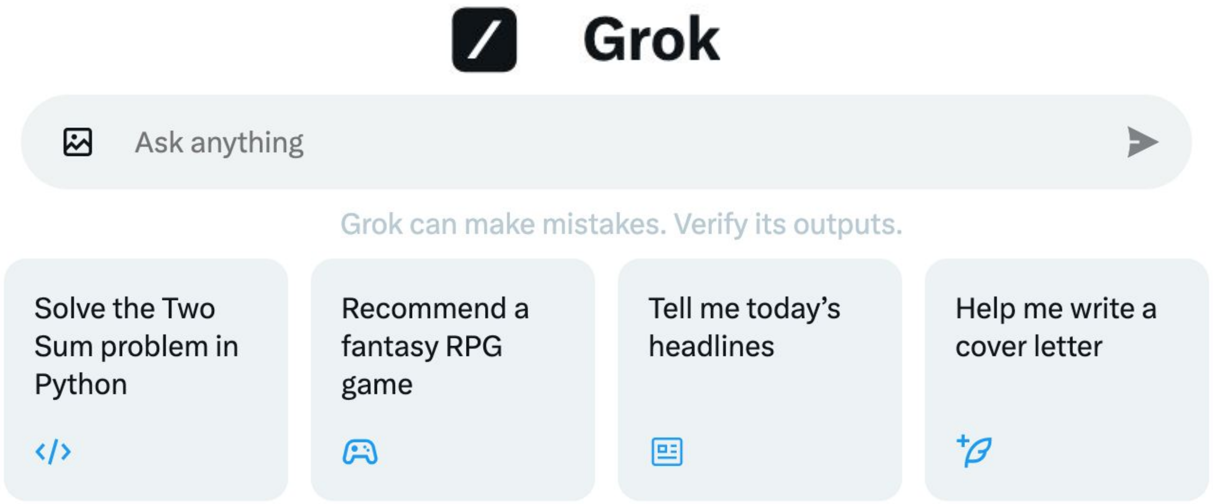
SGLang is a fast serving framework for large language models and vision language models.

What is SGLang?

A **fast inference engine** for LLMs

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

Serves the **production and research** workloads at xAI



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

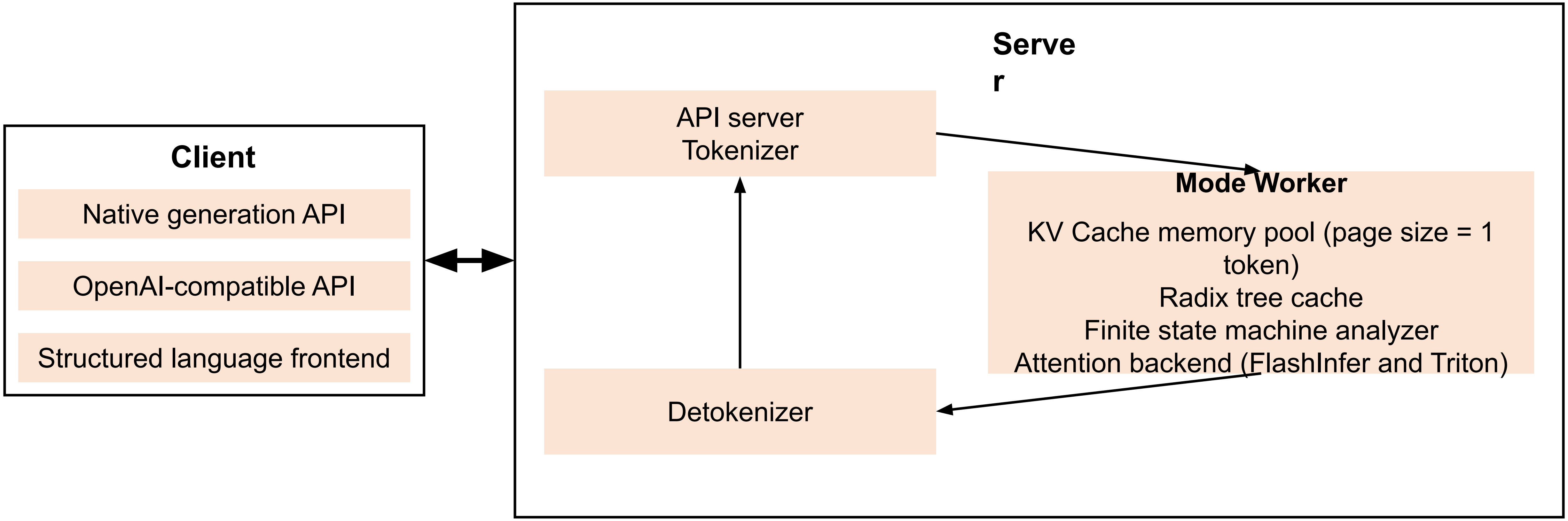
v0.2 (July 2024)

3x higher throughput with low-overhead CPU runtime

v0.3 (Sept. 2024)

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

SGLang architecture overview



Lightweight and customizable code base in Python/PyTorch

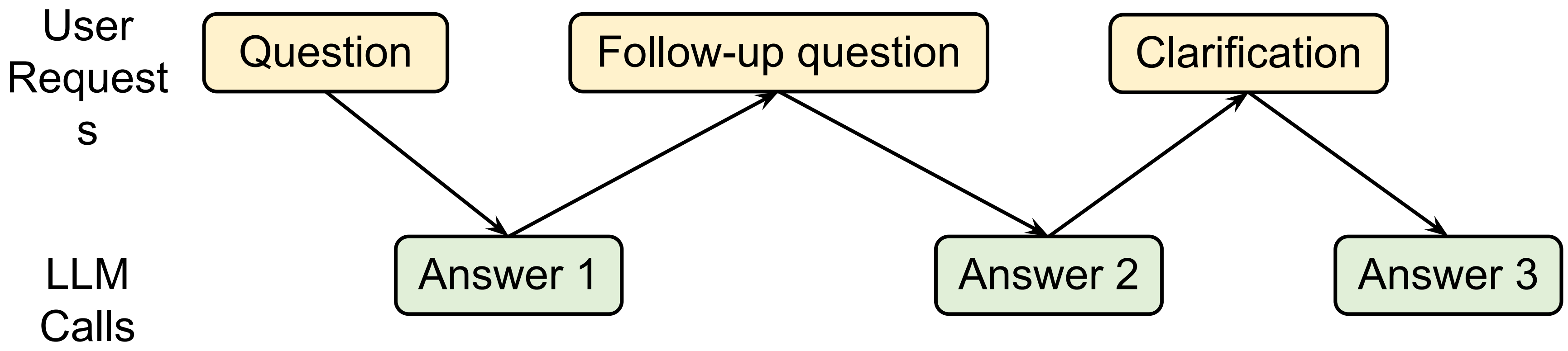
Major Techniques

Four techniques covered in this talk

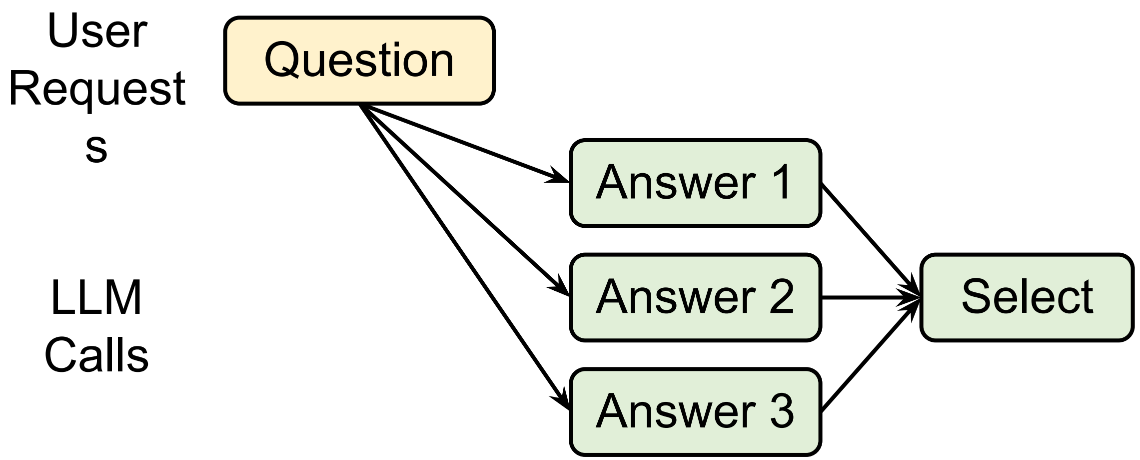
1. Efficient KV cache reuse with RadixAttention
2. Efficient JSON decoding with compressed finite state machine
3. Low-overhead CPU scheduling
4. Torch native optimizations (torch.compile, torchao)

LLM inference pattern:
Complex pipeline with multiple LLM calls

Chained calls



Parallel calls



LLM inference pattern:
Complex pipeline with multiple LLM calls

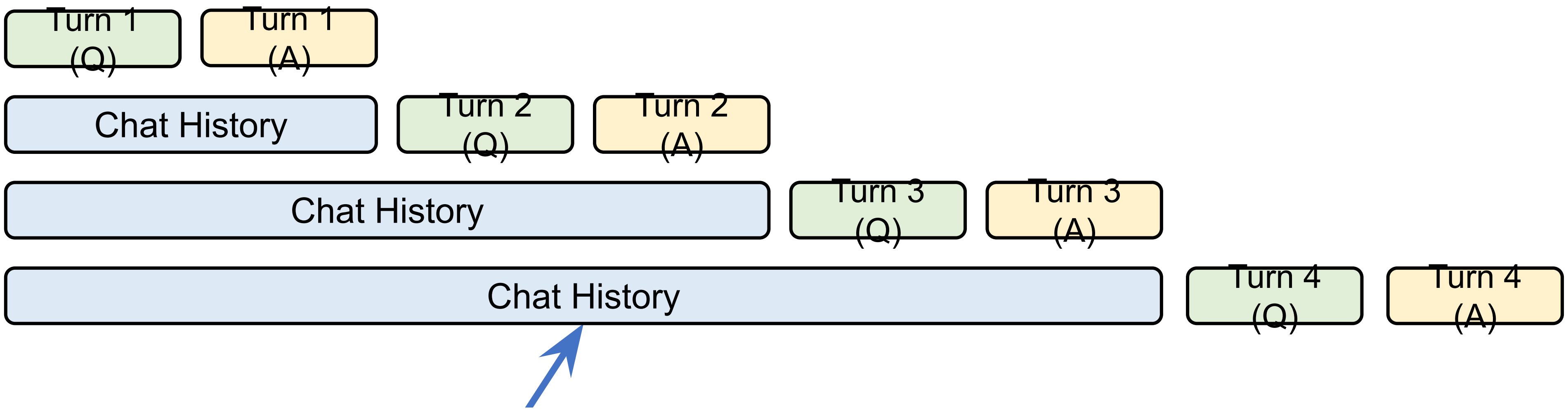
Chained calls

Parallel calls

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

There are rich structures in LLM calls

(a) Multi-turn chat



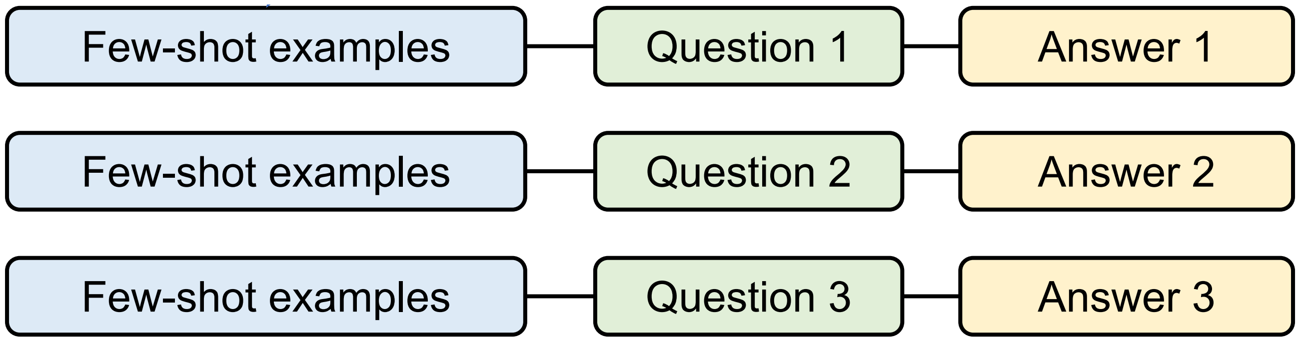
Reusable KV cache (Key-Value cache, some intermediate tensors)

(b) Few-shot learning

Prompt 1

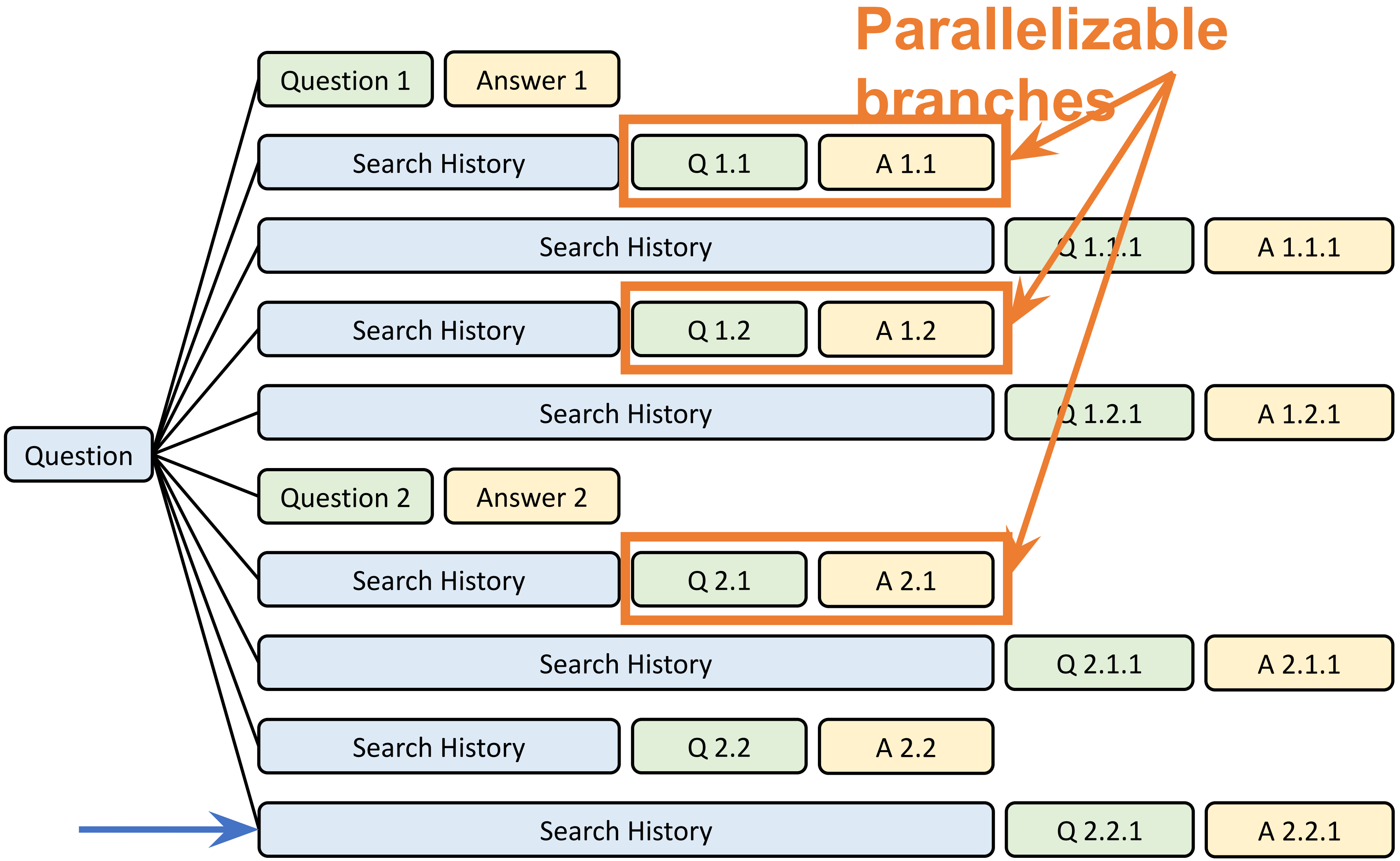
Prompt 2

Prompt 3



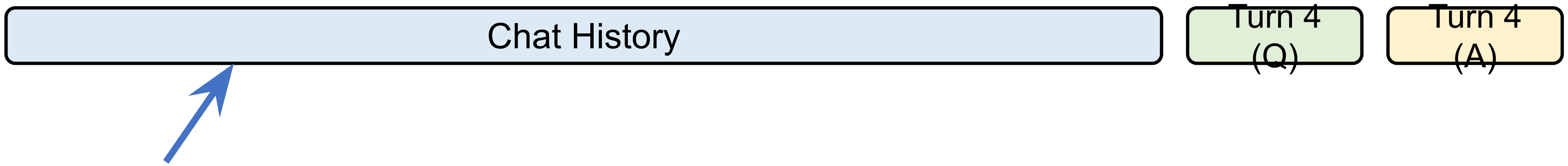
The structures can be very complicated

(c) Tree search with LLM agents



Reusable KV cache

Technique 1: Efficient KV cache reuse with RadixAttention



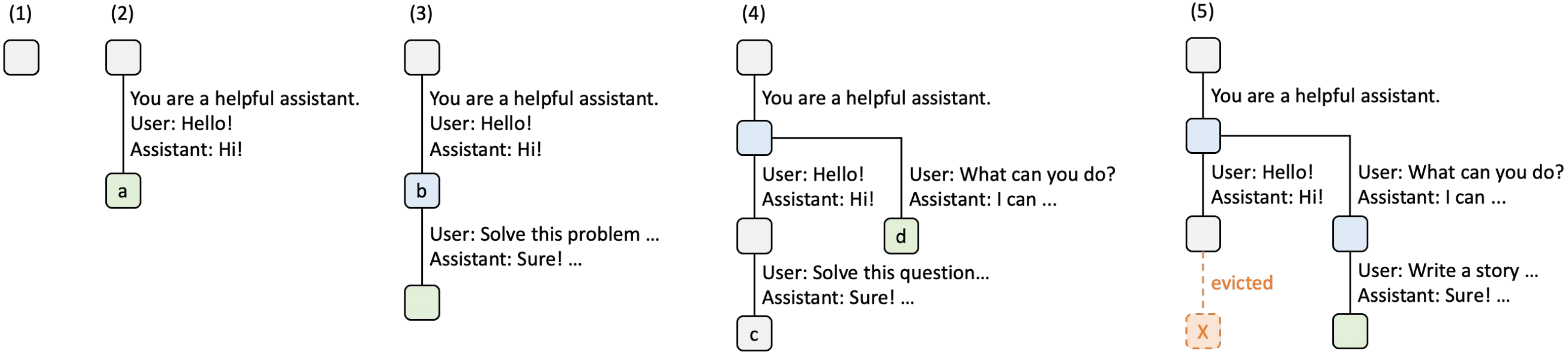
KV Cache

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

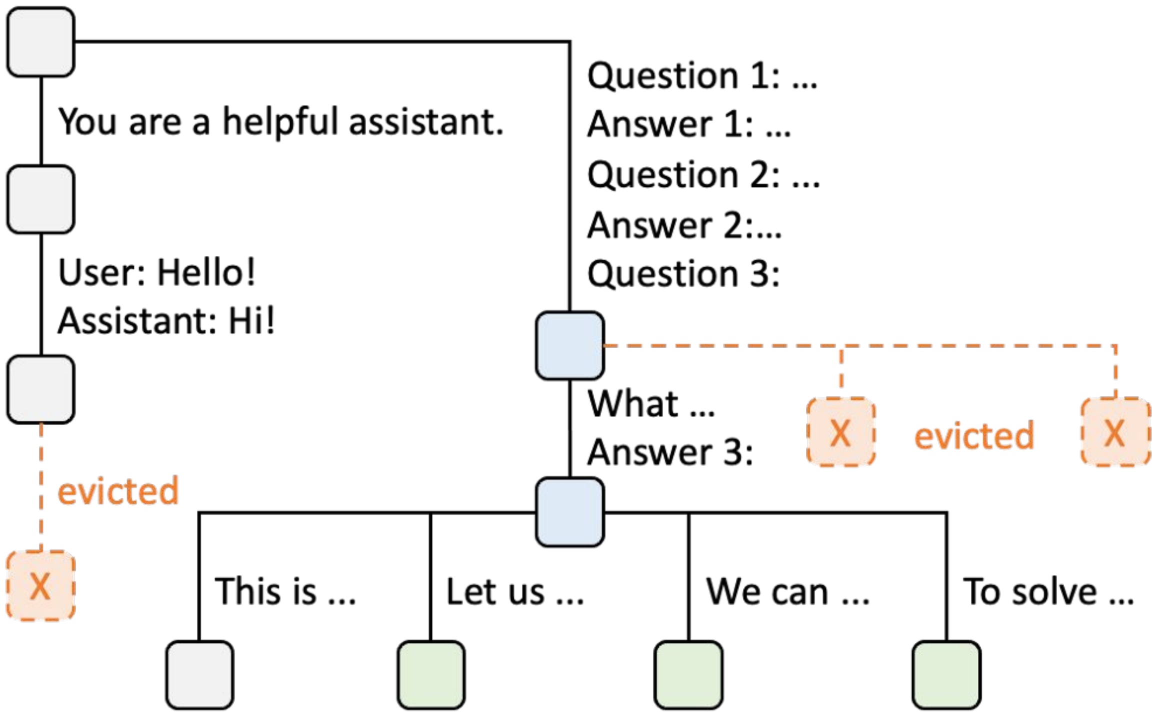
Existing systems: Discard KV cache after an LLM call finishes

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

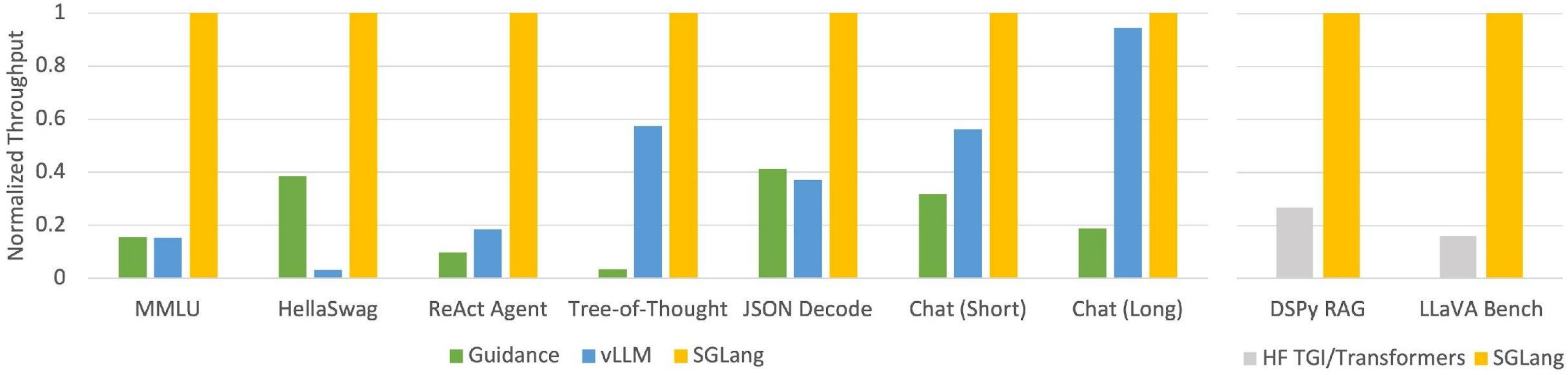
Idea: Utilize **user annotations** and **runtime metrics** for scheduling

Single worker case
Sort the requests in the queue according to matched prefix length

Distributed case
Route the requests to the worker with the matching cache

Results: SGLang is fast and flexible

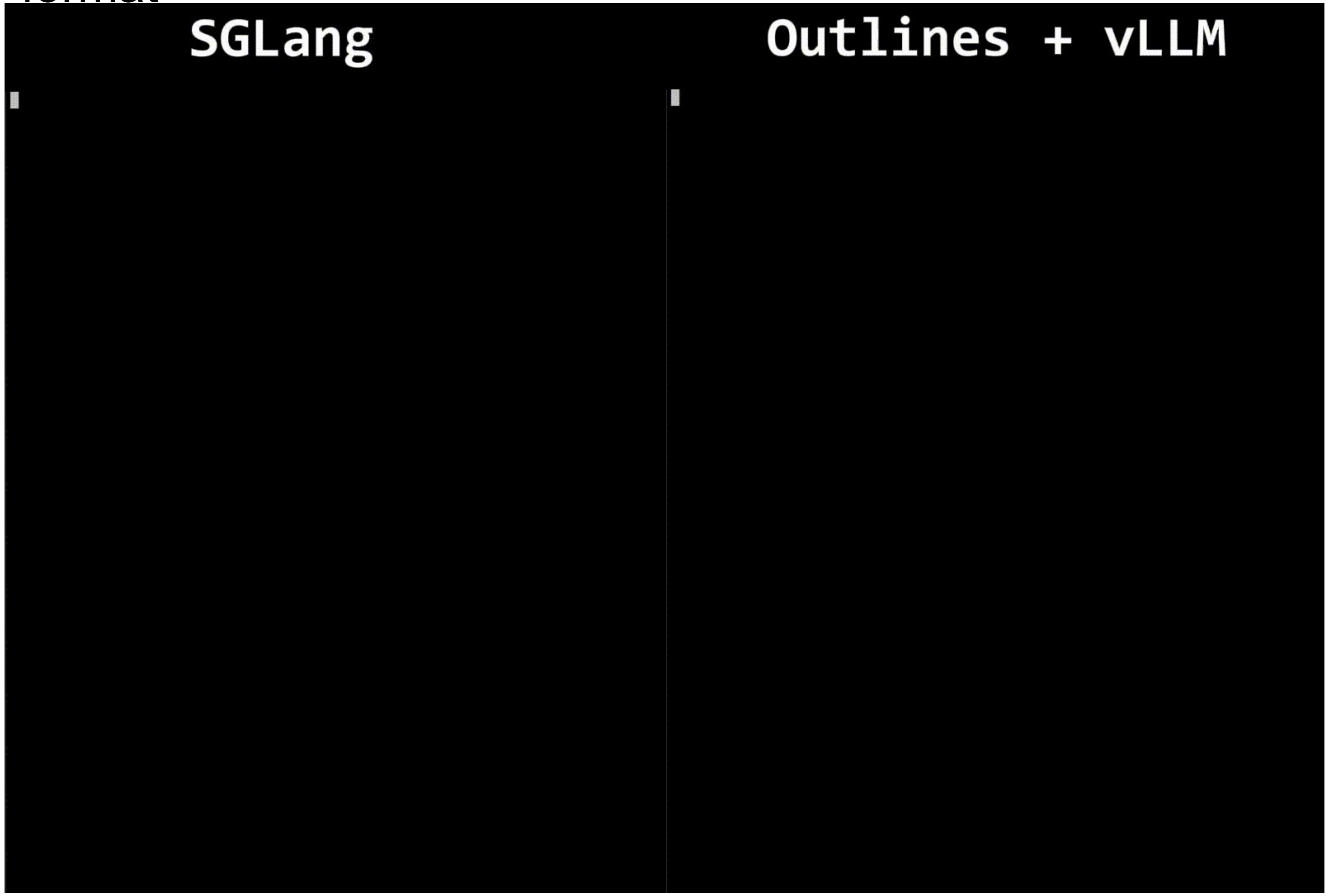
- Up to **5x higher throughput** with KV cache reuse and parallelism
- Works automatically across workloads and text/image tokens



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

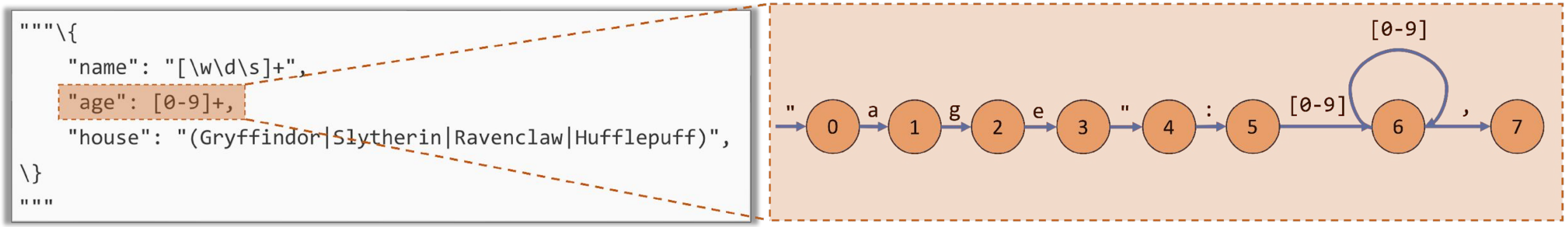
Technique 2: Efficient constrained decoding

Workload: Generate the descriptions of characters in the JSON format



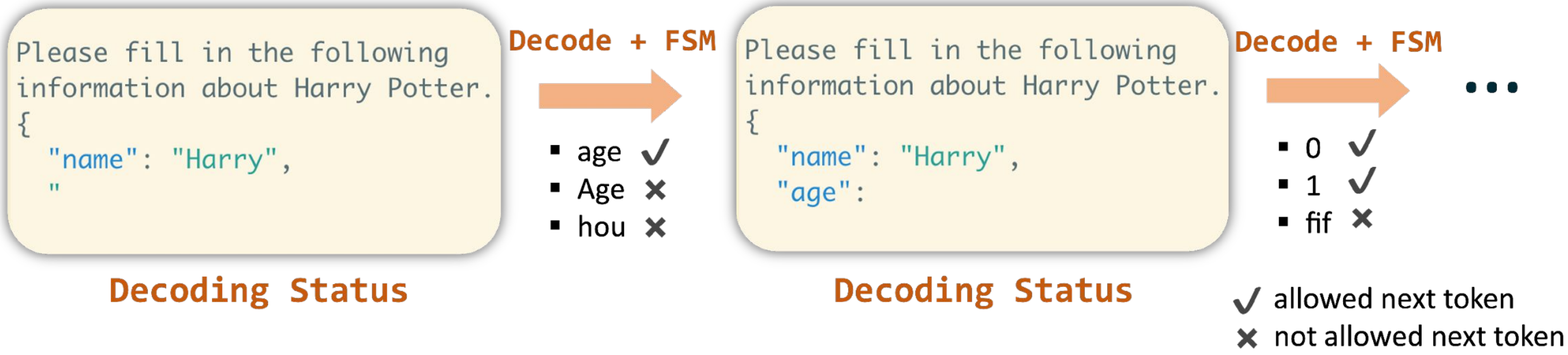
Constrained decoding works by masking the invalid tokens

Constraint decoding: JSON schema -> regular expression -> finite state machine -> logit mask



Regular Expression

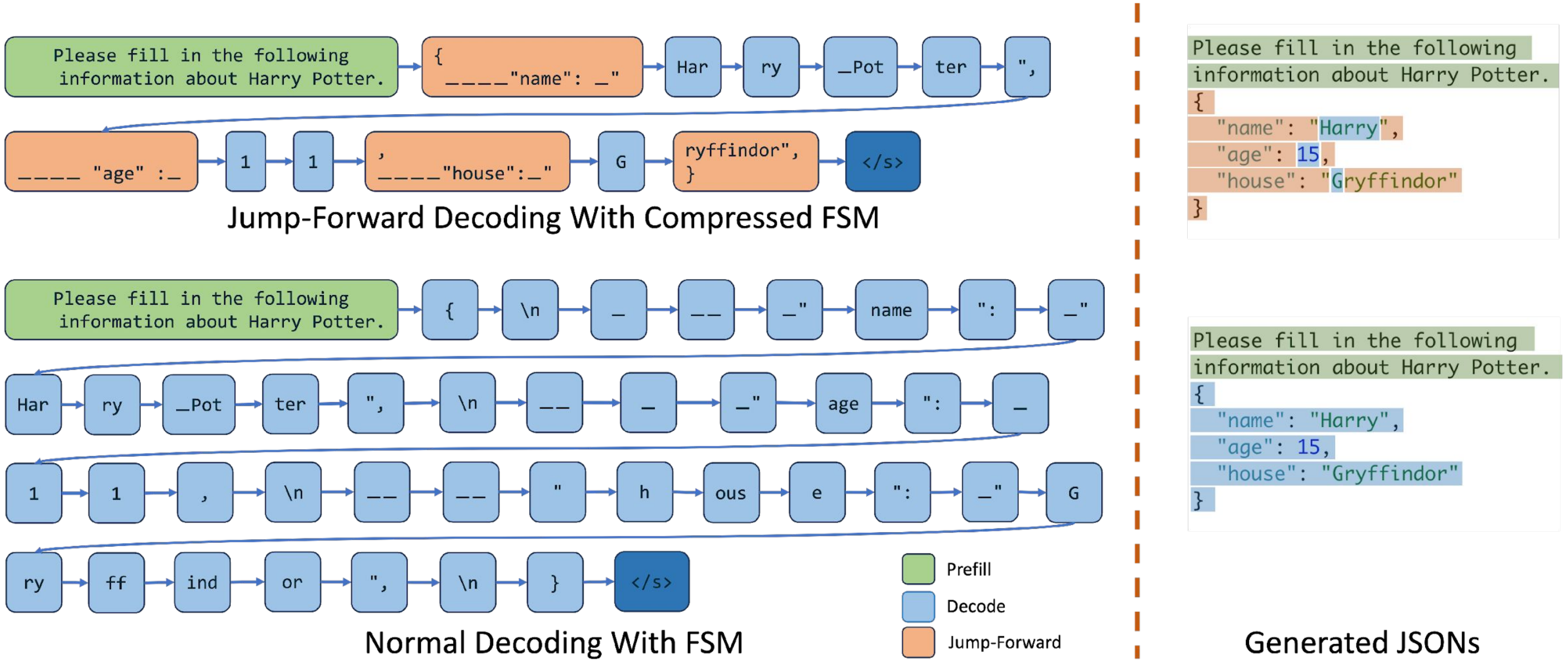
Finite State Machine



Constrained Decoding With Logits Mask

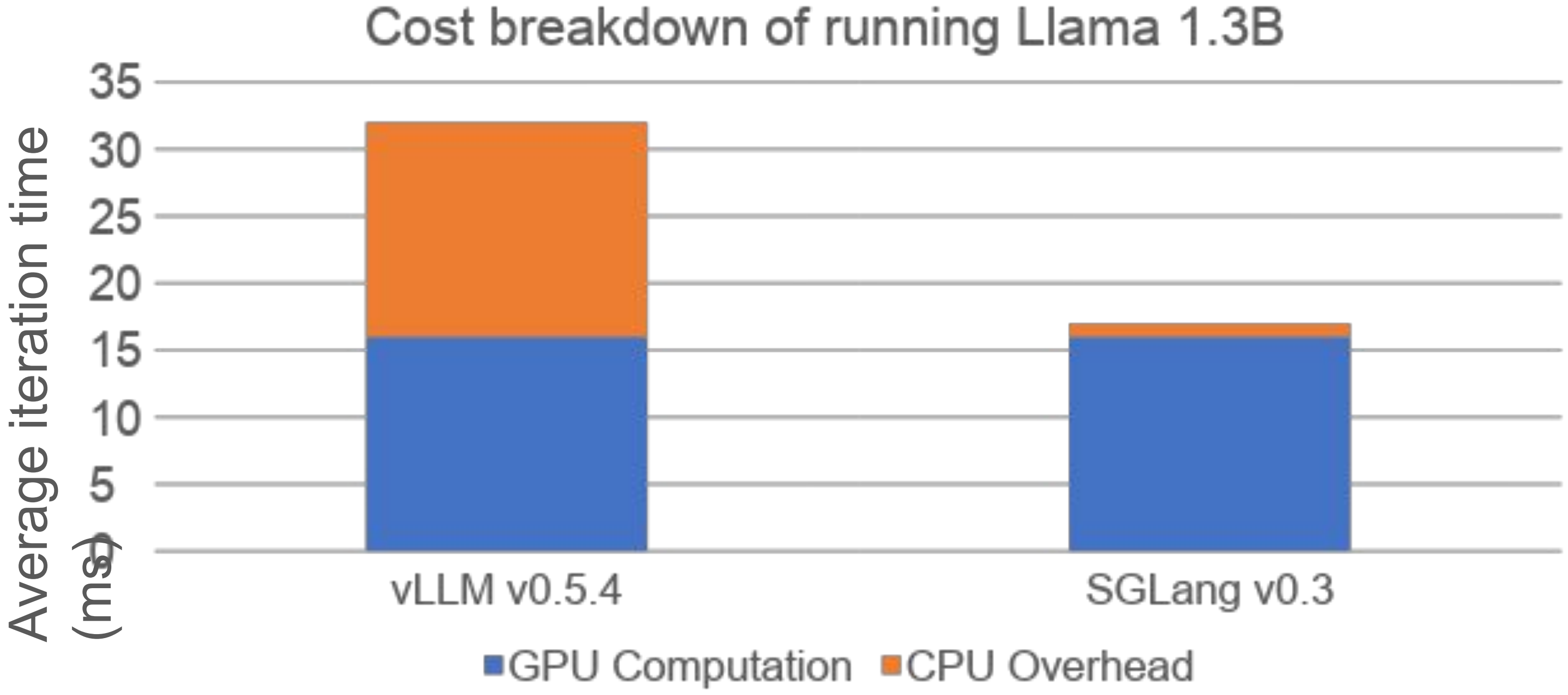
Compressing the finite state machine allows decoding multiple tokens

We can compress many deterministic paths in the state machine



Technique 3: Low overhead CPU scheduling

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

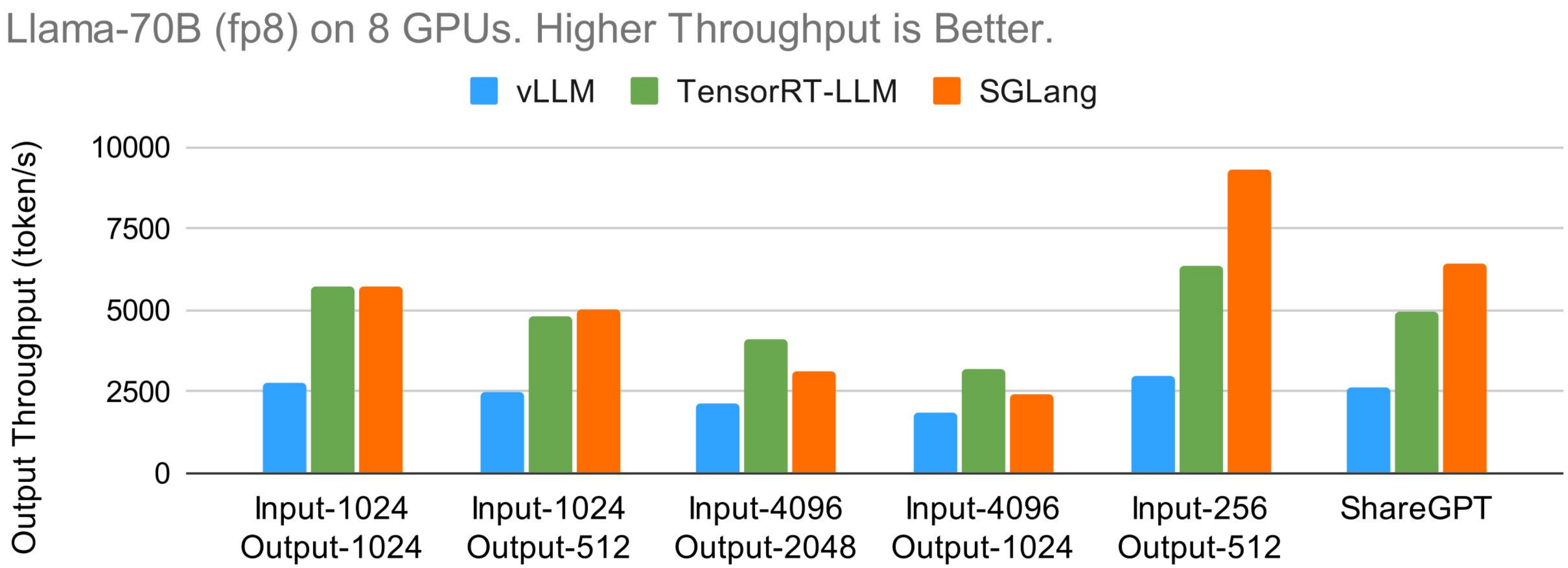


Source: https://mlsys.wuklab.io/posts/scheduling_overhead/

Technique 3: Low overhead CPU scheduling

Idea: Vectorize CPU operations / Overlap CPU scheduling

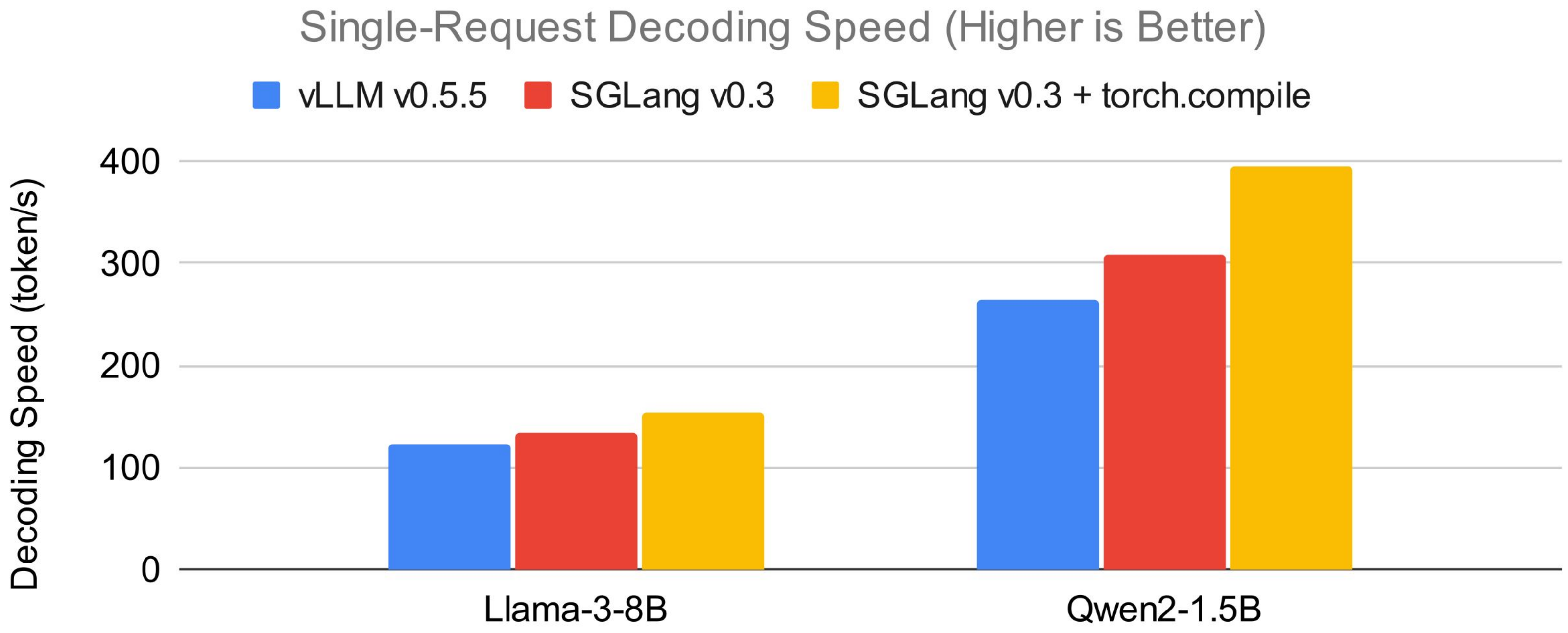
Results: Our python runtime matches C++ runtime and outperforms other python runtime by up to 3x.



Source: SGLang v0.2 blog, https://lmsys.org/blog/2024-07-25-sglang-llama3/

Technique 4: PyTorch-native optimizations

1.5x faster decoding with torch.compile
1.3x faster decoding with torchao int4 quantization (vs. fp8)

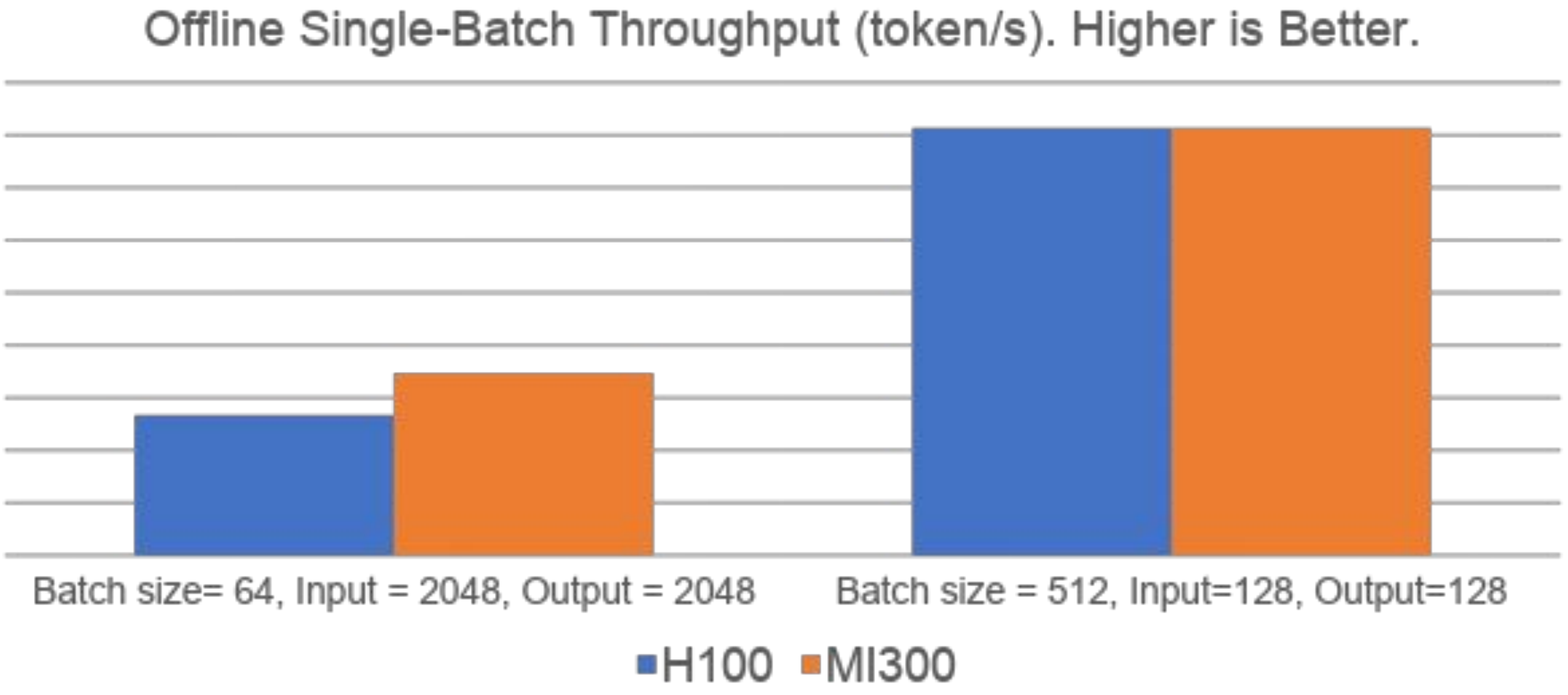


Source: SGLang v0.3 blog, https://lmsys.org/blog/2024-09-04-sglang-v0-3/

Preliminary benchmark results on MI300

Grok-1 (314B MoE, FP8) with SGLang on MI300

Setup: both use the triton attention backend. H100 runs TP=8, MI300 runs 2 x TP=4 thanks to its larger memory.
Preliminary results after one week of optimization. MI300 already shows promising results.



Data and integration contributed by the AMD team

Open-source community and roadmap

Community users and contributors

Roadmap

Performance optimizations

Sequence parallelism and sparse attention for long context inference
Adaptive speculative decoding for all batch sizes
Disaggregated prefill and decoding
Hierarchical radix cache
Faster grammar parsing libraries
Communication and CPU overhead overlapping

Modular design

Integrate PyTorch-native optimizations

Community building

Bi-weekly online development meeting

Acknowledgment

AMD technical support

Xiao
Hai

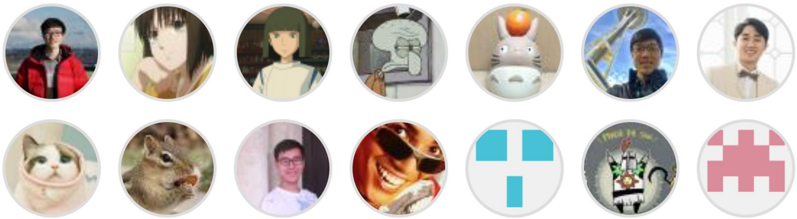
Anush
Elangovan

Soga Lin, Wun-guo Huang

SGLang open-source contributors

Core developers: Ying Sheng, Liangsheng Yin, Yineng Zhang, Ke Bao

Contributors 119



+ 105 contributors

Question & Answer

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

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

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