SGLang v0.2: Faster Interface and Runtime for LLM inference (confidential)

Ying Sheng, Liangsheng Yin, Zhiqiang Xie

Aug -
Dec.
2023

Early Stage: the “programming LLM” paradigm

Jan. -
now
2024

Middle Stage: innovative features and optimizations

now -
2024

Production Stage: research and industry use-cases

Early Stage: the "Programming LLM" Paradigm

From chat and simple prompting to programmatic usage of LLMs

Simple Chat ⇒ LLM Programs

- **Multiple calls**
- Advanced prompting
- Control flow
- Tool use
- External interaction

Existing Systems

Front end language: ignored runtime optimizations

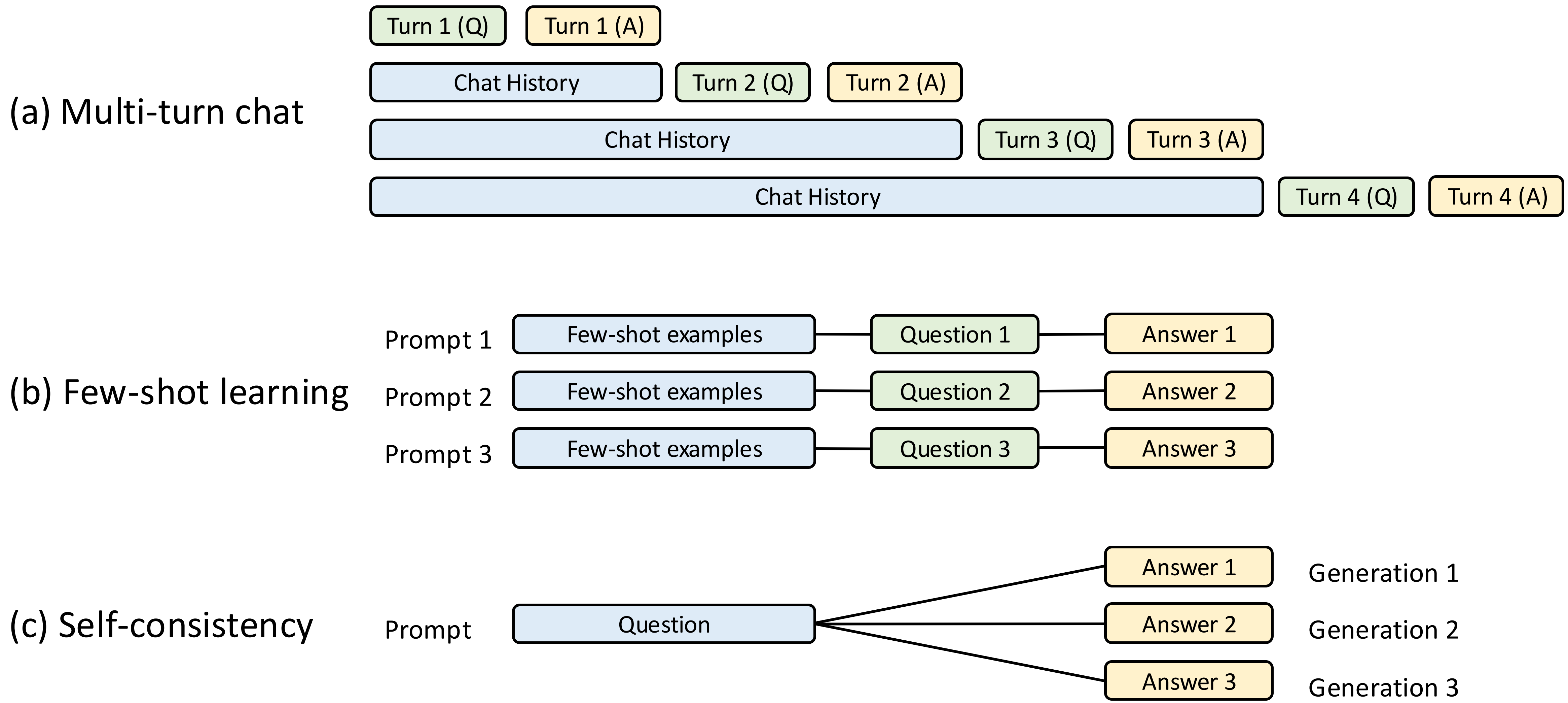
(Guidance, LMQL)

Backend Inference engine: do not know program structure

(NVIDIA TensorRT-LLM, vLLM)

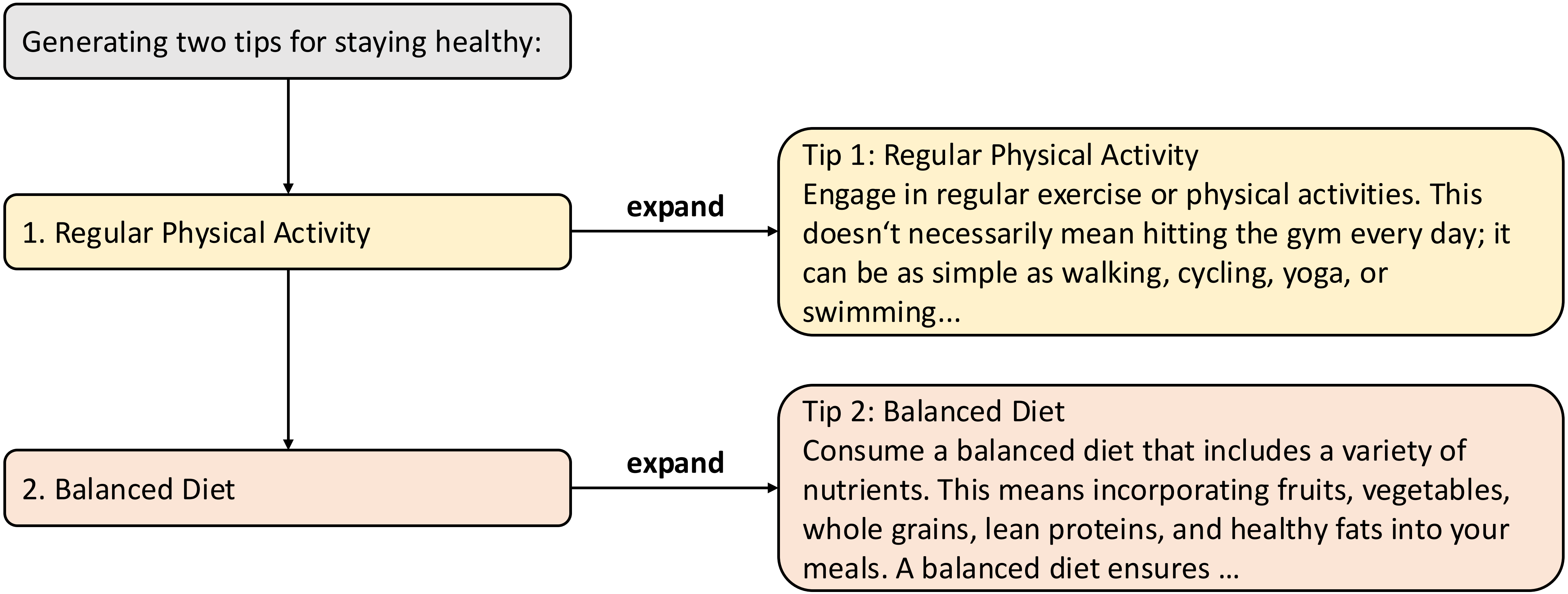
Early Stage

Opportunity: KV Cache Reuse



Early Stage

Opportunity: Parallelism



System Challenges

- How to program these LLM applications?

- How to optimize across multiple LLM calls?

Early Stage

Introducing SGLang: A Structured Generation Language

A “co-design” approach

Front end

- A new domain specific language embedded in Python
- Automatic parallelization and other compiler optimizations

Back end

- Automatic KV cache reuse with **RadixAttention**

Early Stage

API example: A Multi-Dimensional Essay Judge

Frontend

```python
dimensions = ["Clarity", "Originality", "Evidence"]

@function
def essay_judge(s, essay):
  s += "Please evaluate the following essay. " + essay

  # Evaluate an essay from multiple dimensions in parallel
  forks = s.fork(len(dimensions))
  for f, dim in zip(forks, dimensions):
    f += (
      "Evaluate based on the following metric: " +
      dim + ". End your judgement with the word 'END'")
    f += "Judgment: " + f.gen("judgment", stop="END")

  # Merge judgments
  for f, dim in zip(forks, dimensions):
    s += dim + ": " + f["judgment"]

  # Generate a summary and give a score
  s += "In summary," + s.gen("summary")
  s += "I give the essay a letter grade of " +
  s += s.gen("grade", choices=["A", "B", "C", "D"])

ret = essay_judge.run(essay="A long essay ...")
print(ret["grade"])
```

Launch parallel prompts

Non-blocking generation call

Fetching generation results

Constrained generation

Run the function

Early Stage

Compiler Optimizations

- **Building a dataflow graph**
  - Remove Python Interpreter Overhead
  - Global scheduling optimization over the graph
- **Prefetching cached prefixes**
  - Insert prefetching nodes into the graph
- **Code movement for increasing sharable prefix length**
  - Reorder some prompt elements with the help of GPT-4

Frontend

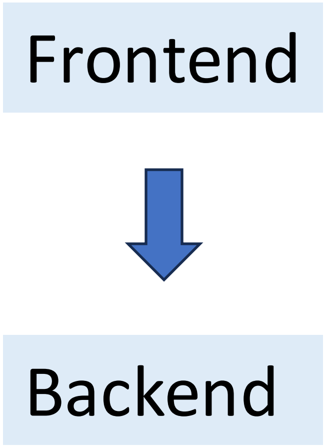
↓

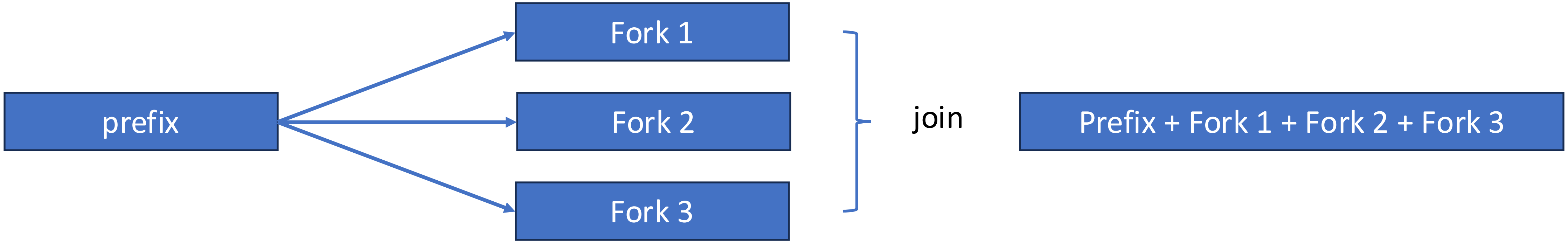
Backend

Early Stage

Prefix caching from request tracking?

- In multi-turn chat, retrieval tasks, etc
  - The interpreter tracks the request id (rid) and caches the history before it ends.
  - Only needs to match the rid.
  - “pin” is a primitive of fixing a prefix to be cached.
  - “fork/join” primitives





Cannot reuse shared prefix across requests!

Aug - Dec. 2023

Early Stage: the “programming LLM” paradigm

Jan. - now 2024

Middle Stage: innovative features and optimizations

Focused efforts on backend/runtime performance

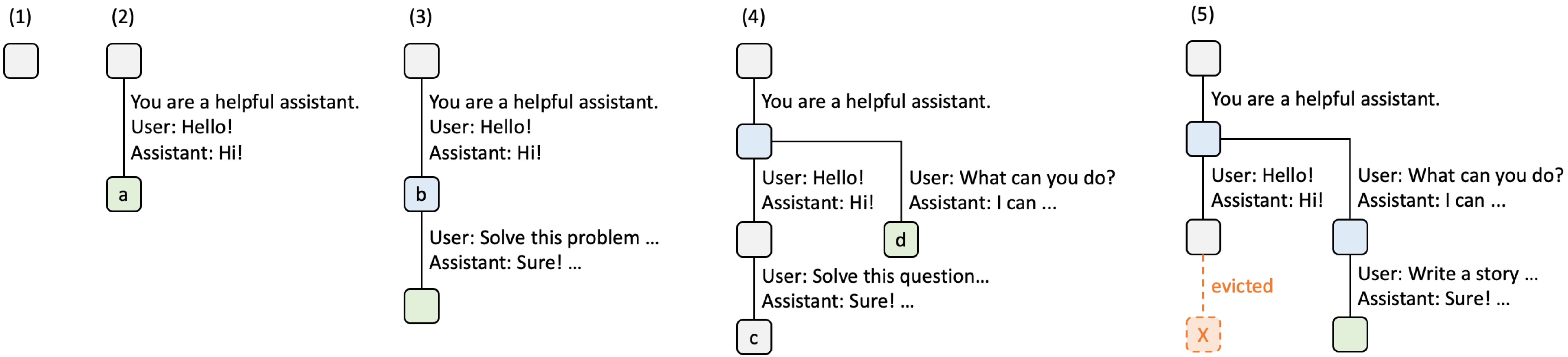
now - 2024

Production Stage: research and industry use-cases

Runtime (SRT) with RadixAttention

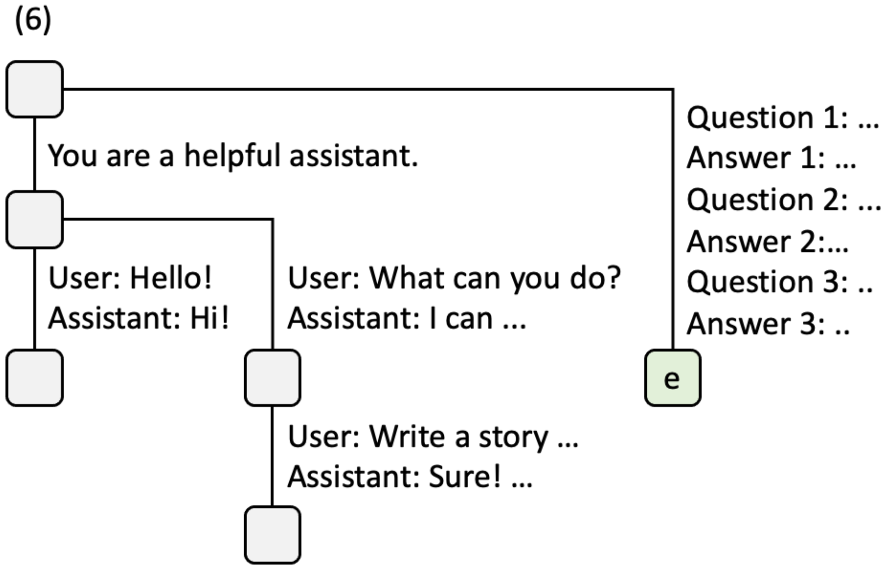
Existing Systems: Discard KV cache after a request finishes.

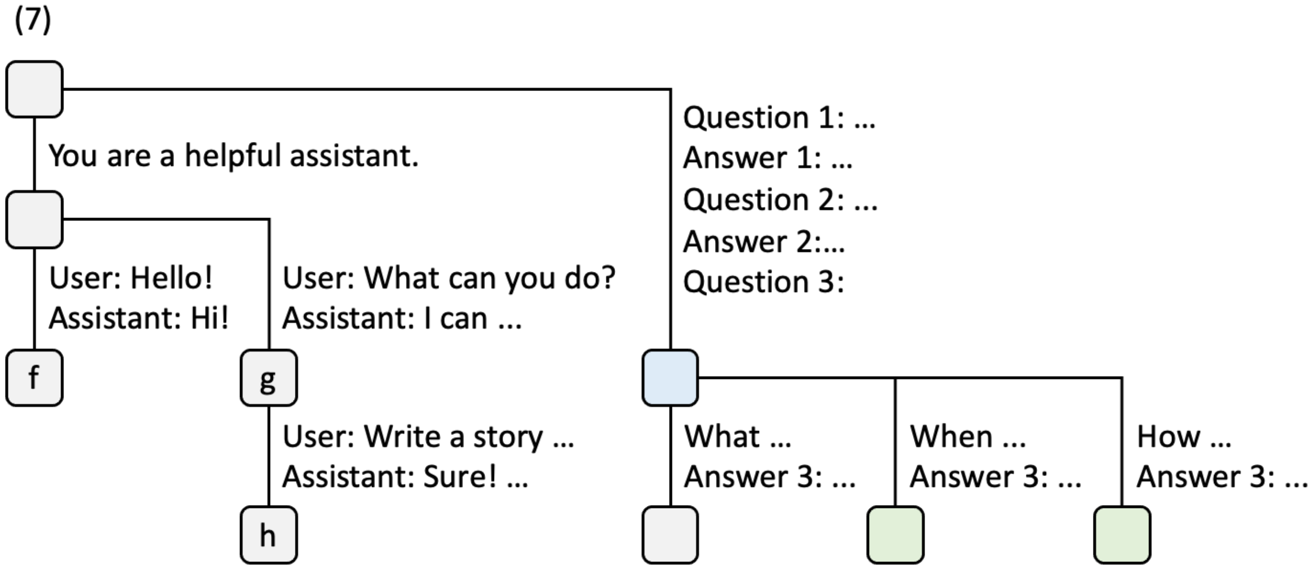
Ours: Maintain an LRU cache of the KV cache of all requests in a radix tree (compact prefix tree).



Runtime (SRT) with RadixAttention

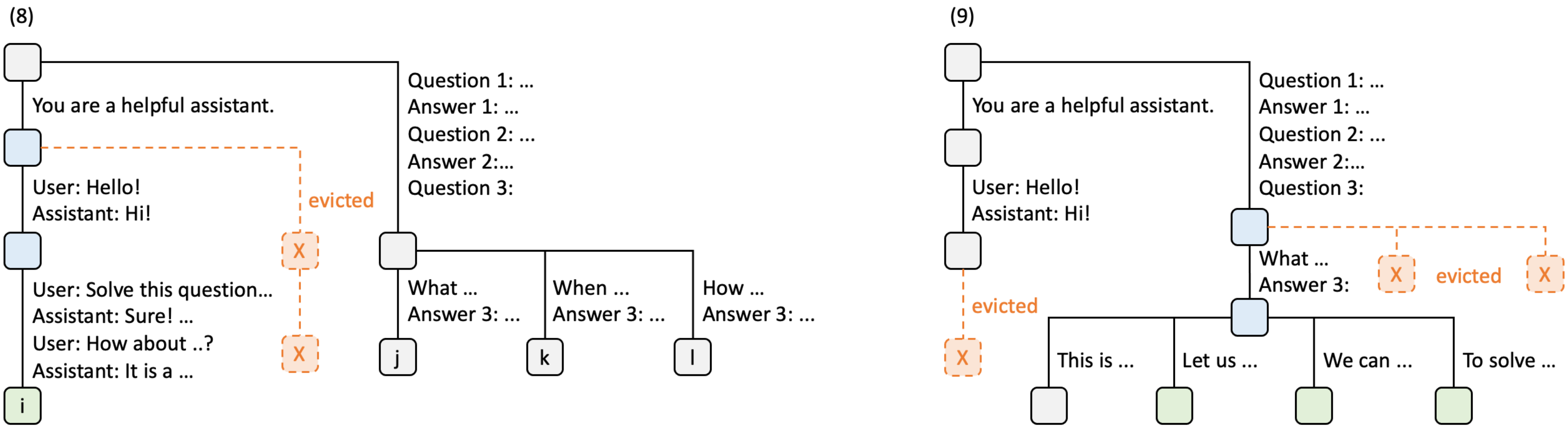
Maintain an LRU cache of the KV cache of all requests in a radix tree.





Runtime (SRT) with RadixAttention

Maintain an LRU cache of the KV cache of all requests in a radix tree.



Cache Aware Scheduling

- In the request queue, sort the requests according to the matched prefix length
  - Achieves good cache hit rate

- Future work
  - Distributed cache aware scheduling for multiple data parallel workers
  - Fairness to prevent starvation (https://arxiv.org/abs/2401.00588)

Aug - Dec. 2023

Early Stage: the “programming LLM” paradigm

Jan. - now 2024

Middle Stage: innovative features and optimizations

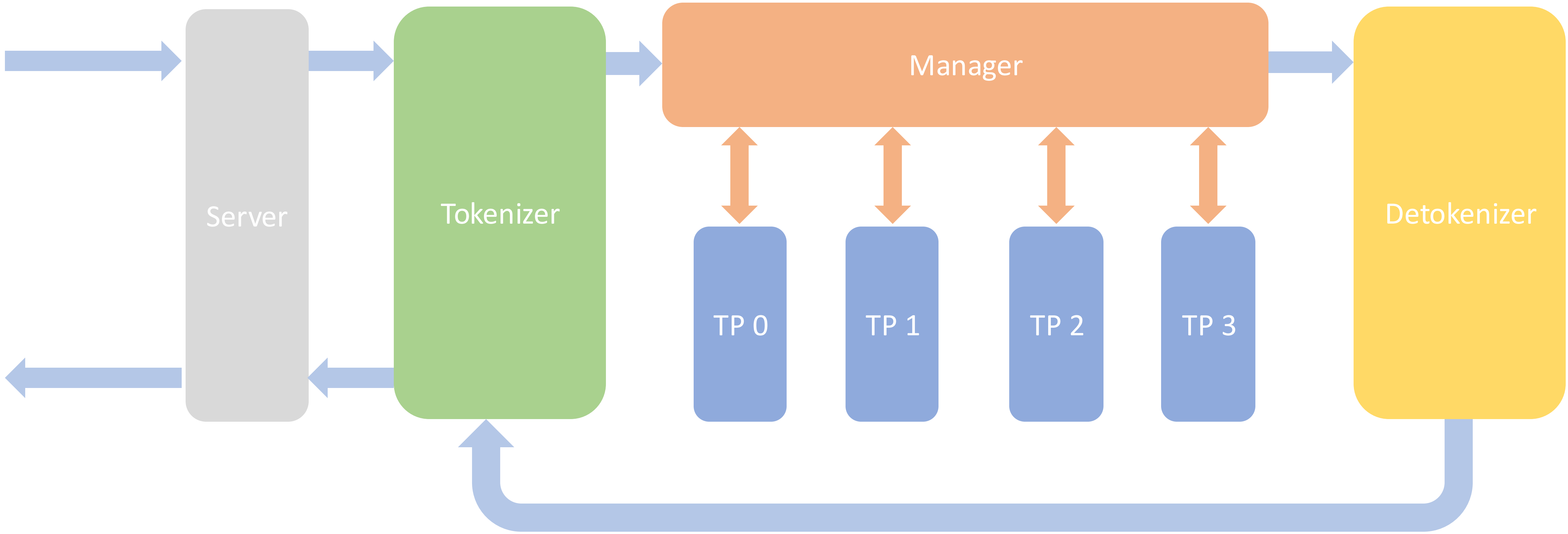
RadixAttention

Upper-level Scheduling

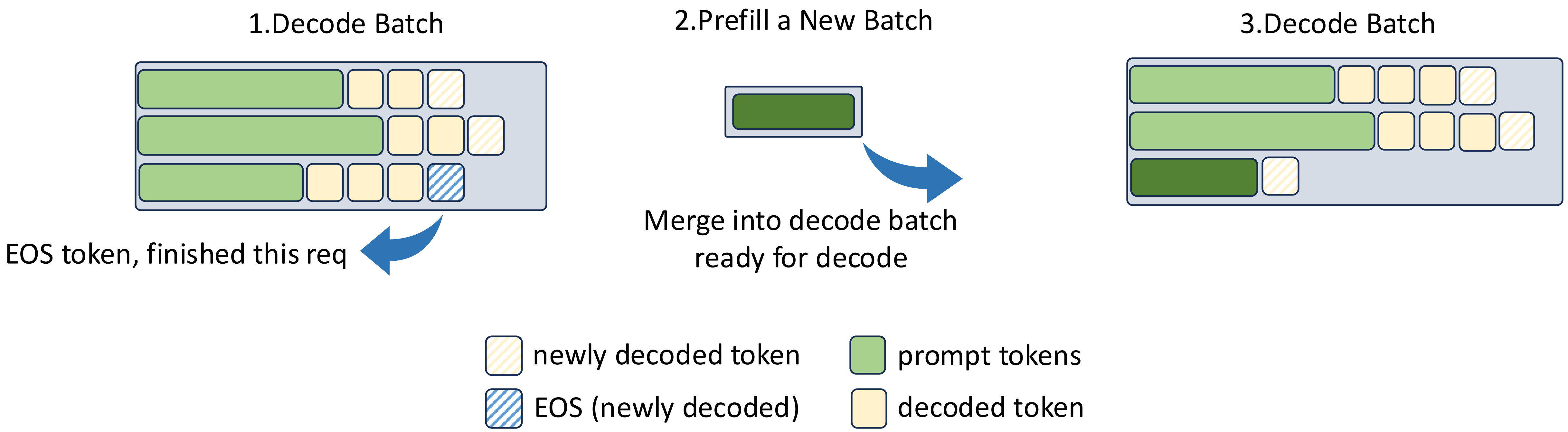
now - 2024

Production Stage: research and industry use-cases

SGLang Structure: Pipeline

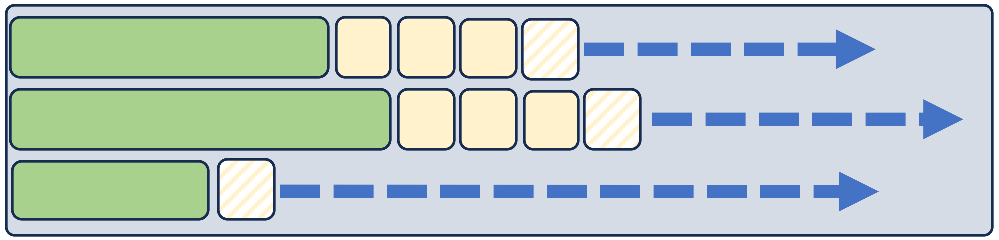


SGLang Structure: Inside TP Worker



How to always keep the batch size large enough?

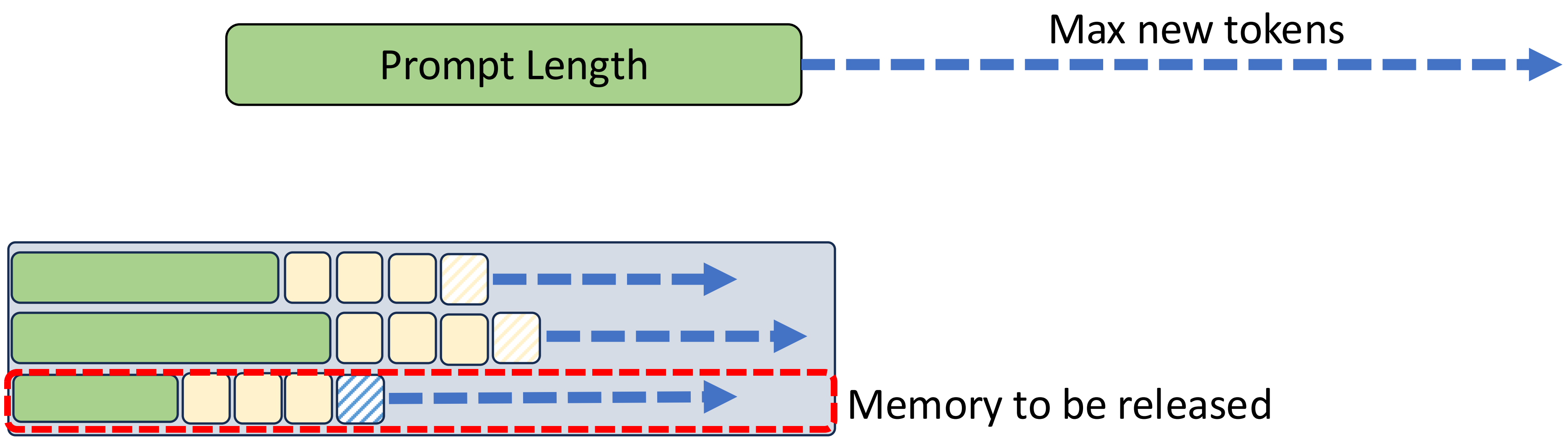
Dynamically Adjust the new token ratio estimation



The max context length
decided by max new tokens

- There is a lot of space left in the GPU memory
- We do not need to reserve every token in max new tokens

Dynamically Adjust the new token ratio estimation



1. The EOS would be earlier than the max new tokens.
2. There are always requests finished and release all the memory.

Only preserve $\beta\ \times$ max_new_token tokens in advance, and adjust β dynamically.

Aug - Dec. 2023

Early Stage: the “programming LLM” paradigm

Jan. - now 2024

Middle Stage: innovative features and optimizations

RadixAttention

Upper-level Scheduling

Jump-forward decoding

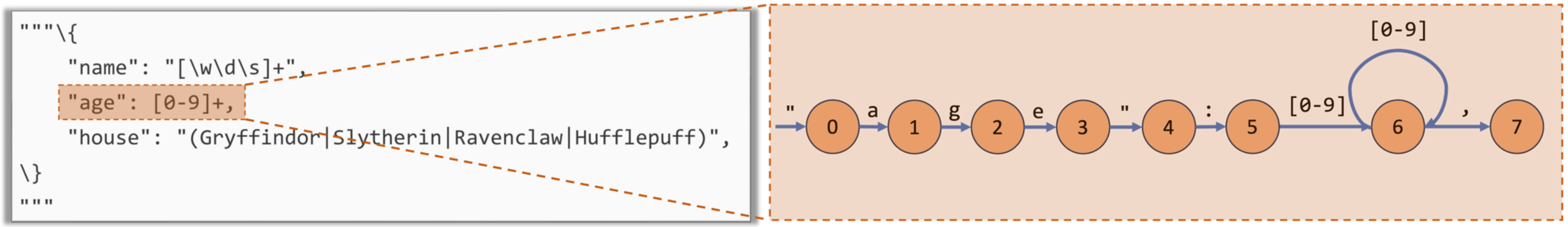
now - 2024

Production Stage: research and industry use-cases

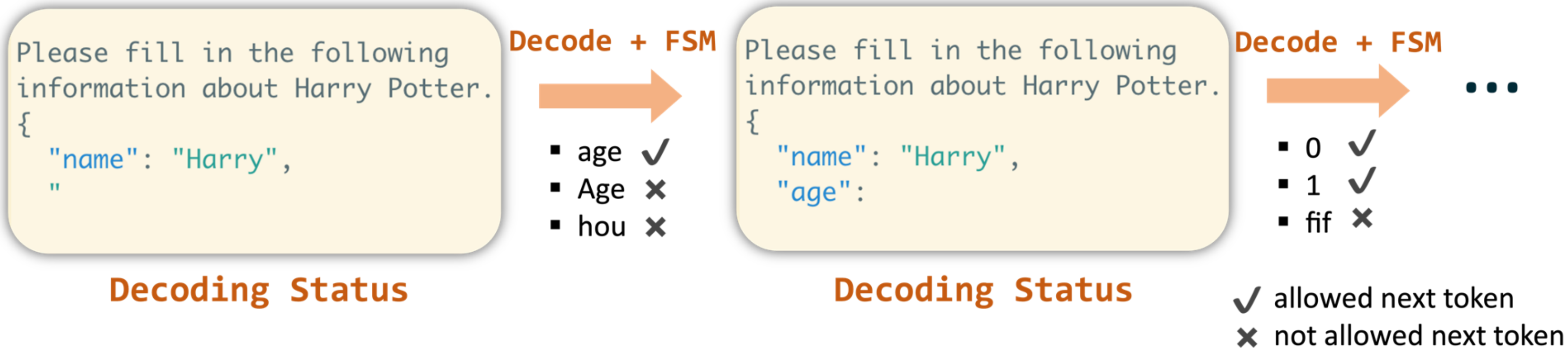
Jump-forward JSON Decoding

Method

- Analyze the regular expression
- Compress the finite state machine
- Decode multiple tokens at the same time

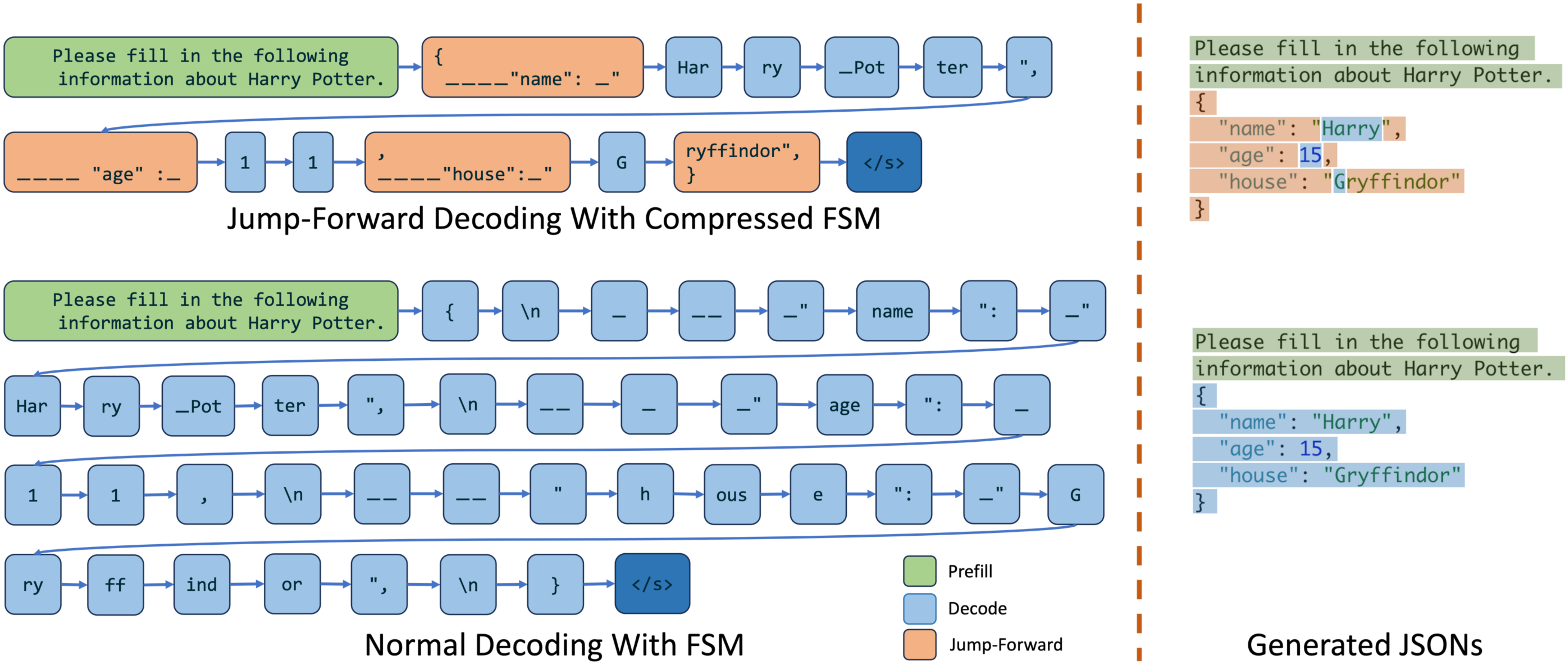


Regular Expression Finite State Machine



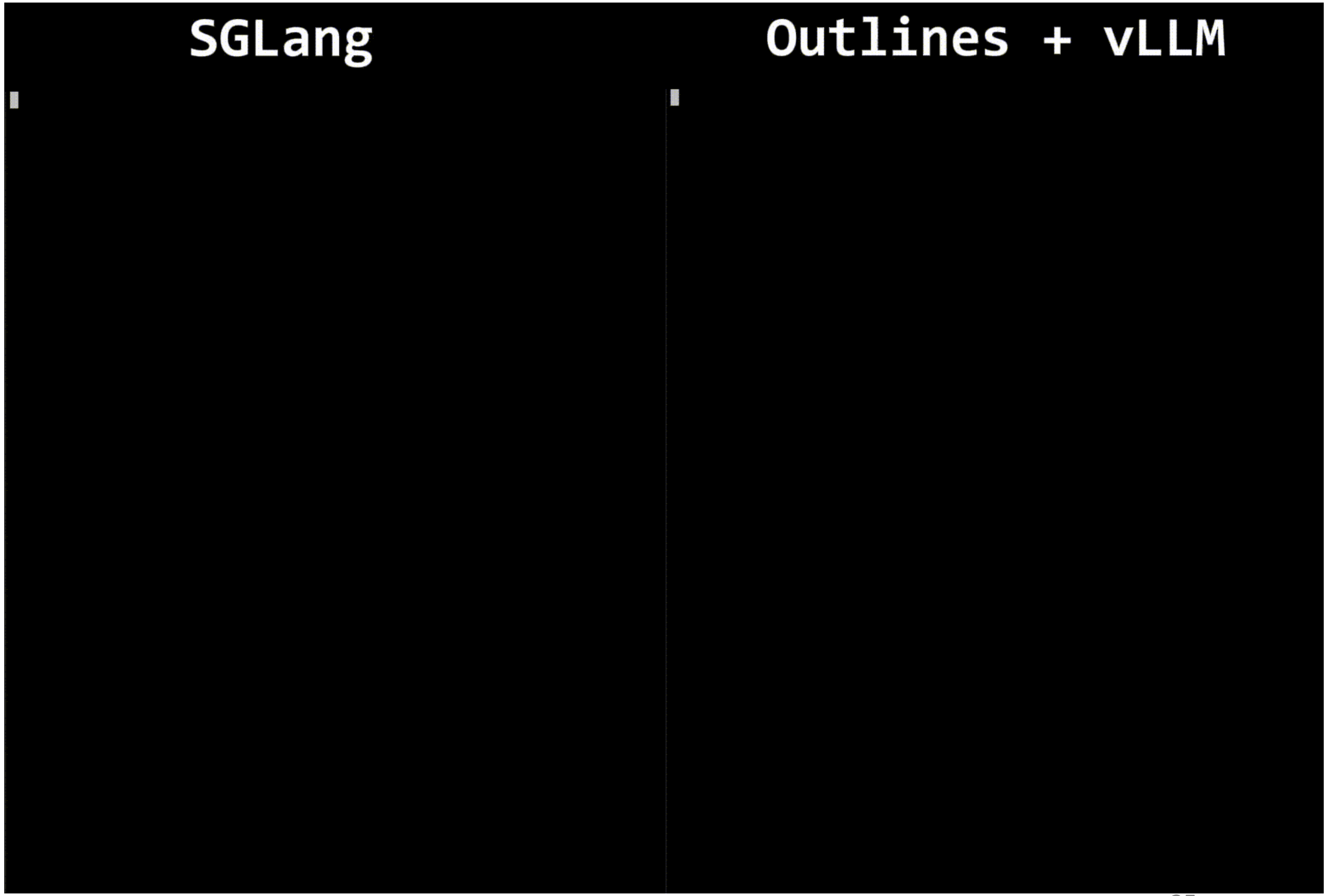
Constrained Decoding With Logits Mask

Speedup Regex Guided Generation



Jump-forward JSON Decoding

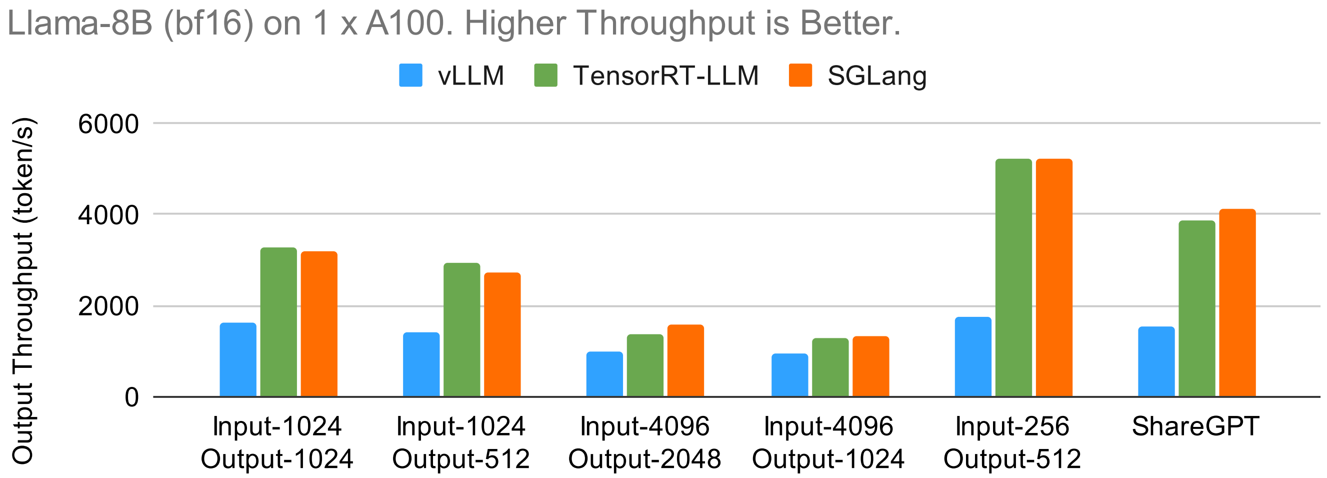
Results:
3x faster latency
2.5x higher throughput

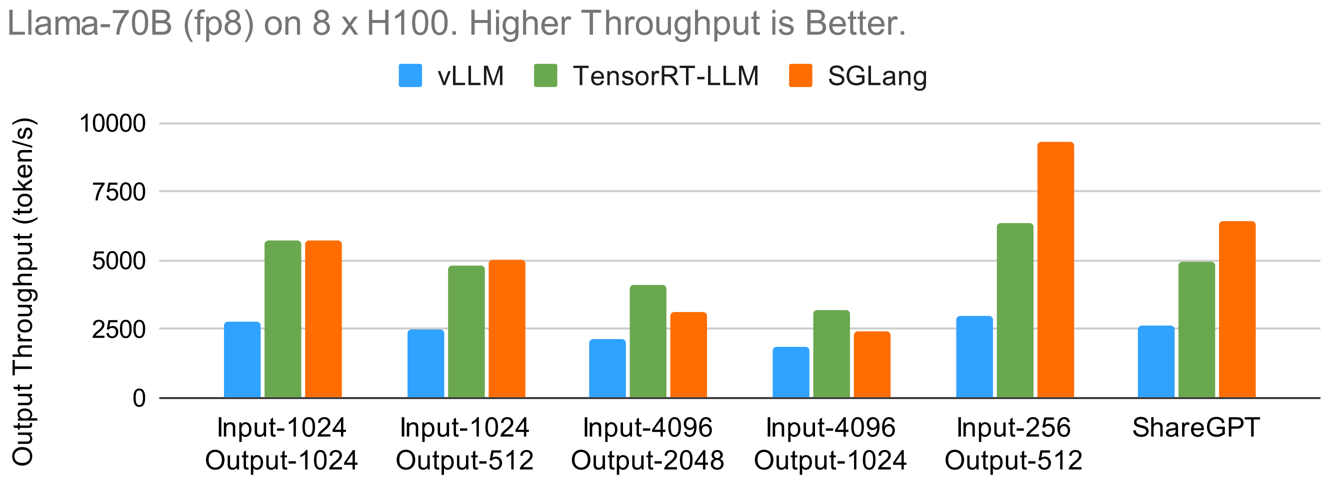


Summary: techniques in SGLang

- RadixAttention
- Jump-forward JSON Decoding
- Torch Compile
- Flashinfer Kernels
- Chunked Prefill
- Continuous Batching
- Token Attention(Paged Attention with page_size = 1)
- CUDA Graph
- Interleave window attention

SGLang v0.2 Results





Aug - Dec. 2023

Early Stage: the “programming LLM” paradigm

Jan. - now 2024

Middle Stage: innovative features and optimizations

now - 2024

Production Stage: research and industry use-cases

Research and industry use cases

x.ai: Production serving of grok-2 and grok-2-mini on X

Databricks: accelerate research workflow by 3.3x

LMSys Chatbot Arena: serving vision language models

LLaVA OneVision: serving multi-modal image and video models

ByteDance

Future work

- multi-level cache
- distributed radix attention for replicas
- long-context sequence parallelism
- speculative decoding
- communication overlapping
- ……

Do the serving engines come to converge on performance?

YES and NO

Basic performance eventually converge

But there are more sophisticated workloads from different scenarios:
RAG systems, agent systems, …

We never forget about the origin of SGLang!
Structured inputs, interactions with different resources, multi-modality, …

Principles in future development

Simplism

Minimalism

Modularity

Ease of use

Development velocity

Performance

Future work

- multi-level cache
- distributed radix attention for replicas
- long-context sequence parallelism
- speculative decoding
- communication overlapping
- ……

Context KV caching is huge for serving!

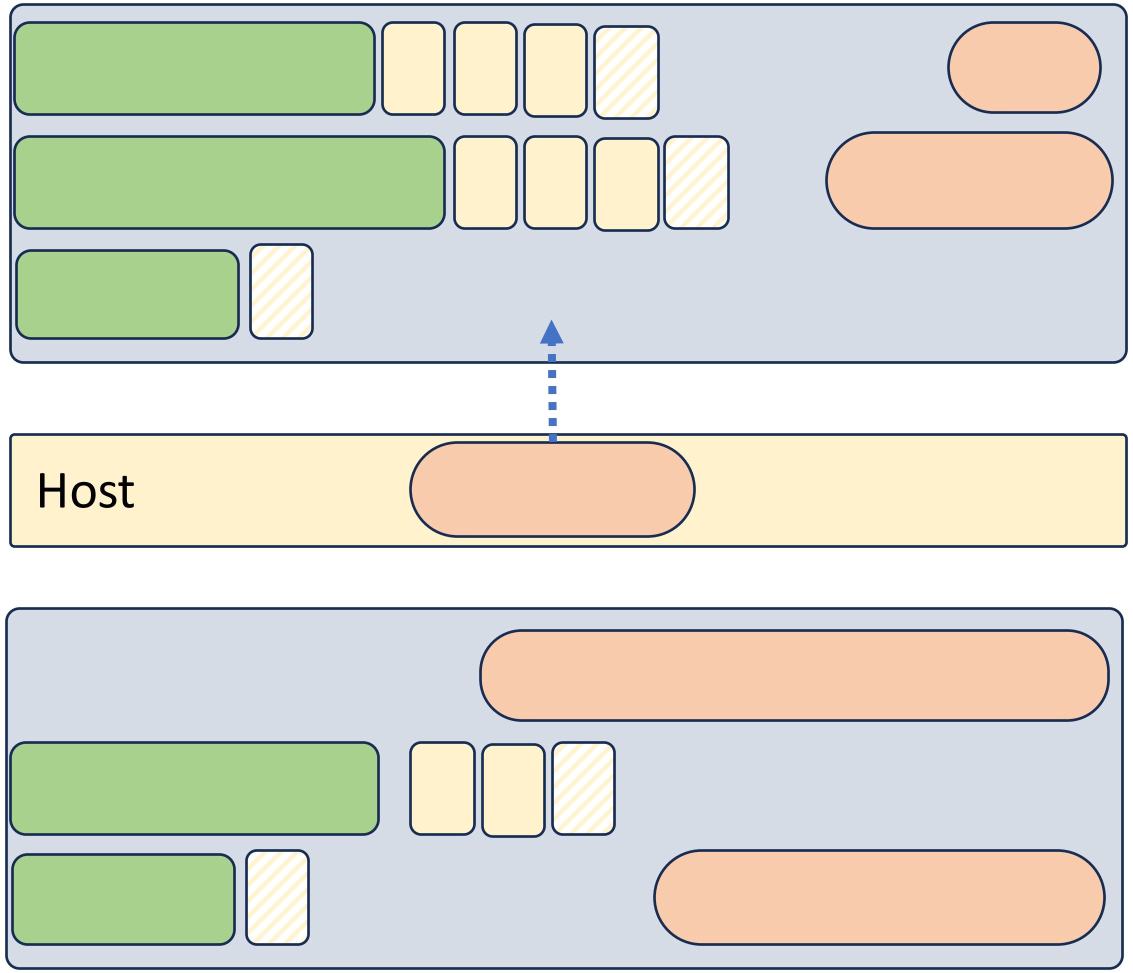
“Our system achieves a **95% cache rate**, further reducing inference cost.”
-- from Character AI

“Even without any optimization, historical data shows that users **save over 50% on average**.”
-- from DeepSeek

“We observed a **52.4% cache hit** rate for LLaVA-Next-34B and **74.1%** for Vicuna-33B.”
-- from Chatbot Arena

…

Challenges for caching

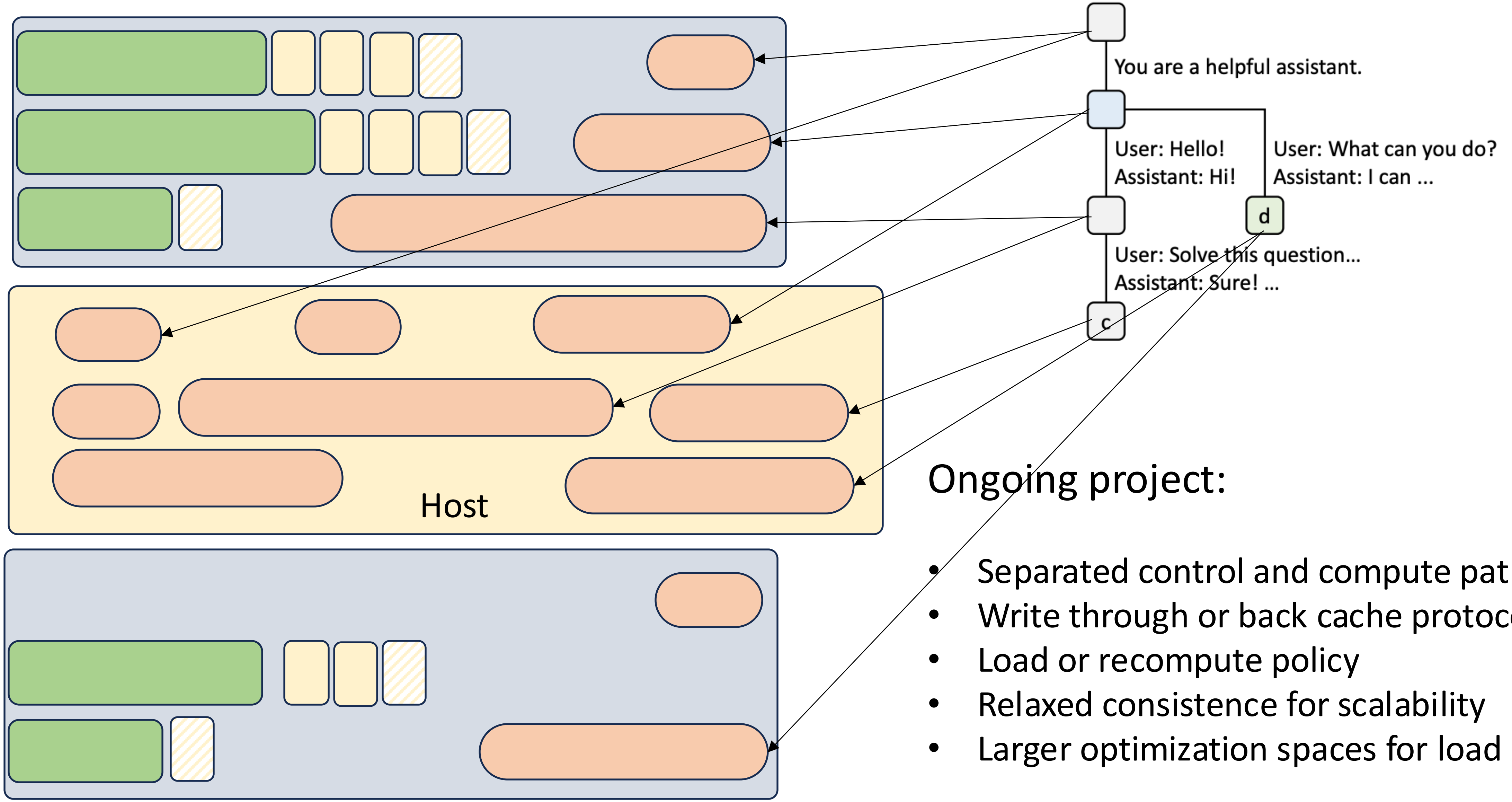


Historical KV caches compete GPU memory with ongoing requests

Offloading and reloading requires more user effort and potentially increase latency

Trade-off between load balancing and cache hit rate for multi-GPU scenarios

Towards a transparent multi-level caching



Ongoing project:

- Separated control and compute path
- Write through or back cache protocol
- Load or recompute policy
- Relaxed consistence for scalability
- Larger optimization spaces for load balancing

Design principle: separated control and compute path