SGLang

INTRODUCTION TO LLM
INFERENCE SERVING SYSTEMS

CHUHONG YUAN

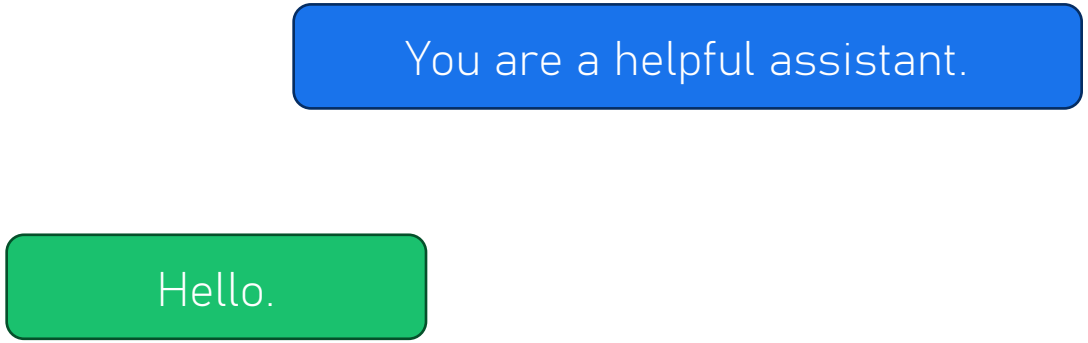
Motivation

- Language model (LM) programs
- Programming LM programs is tedious and difficult
- Redundant computation and memory usage
  - KV Cache reuse opportunities
  - Constrained decoding for structured outputs

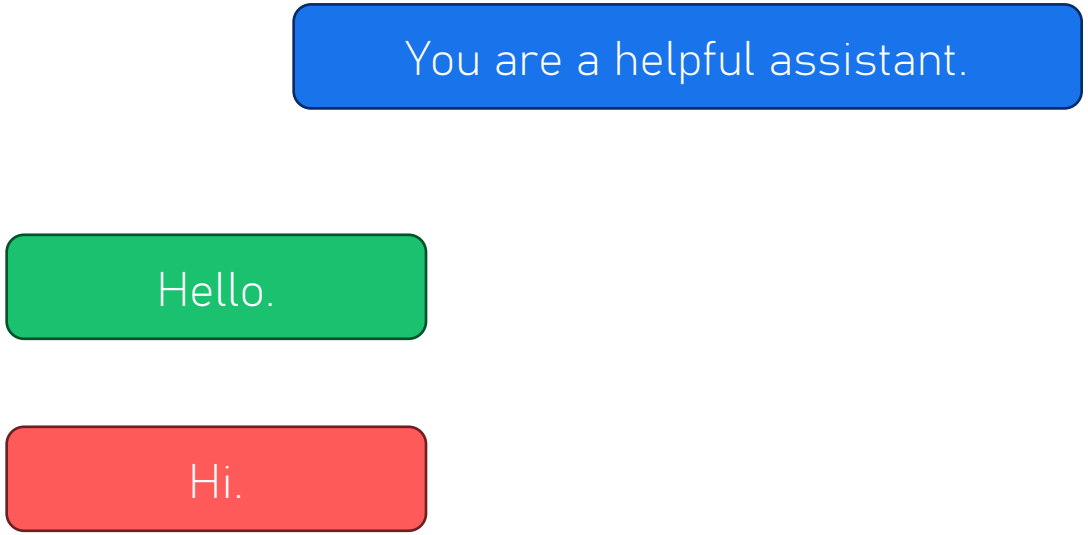
Duplicated Prefixes

You are a helpful assistant.

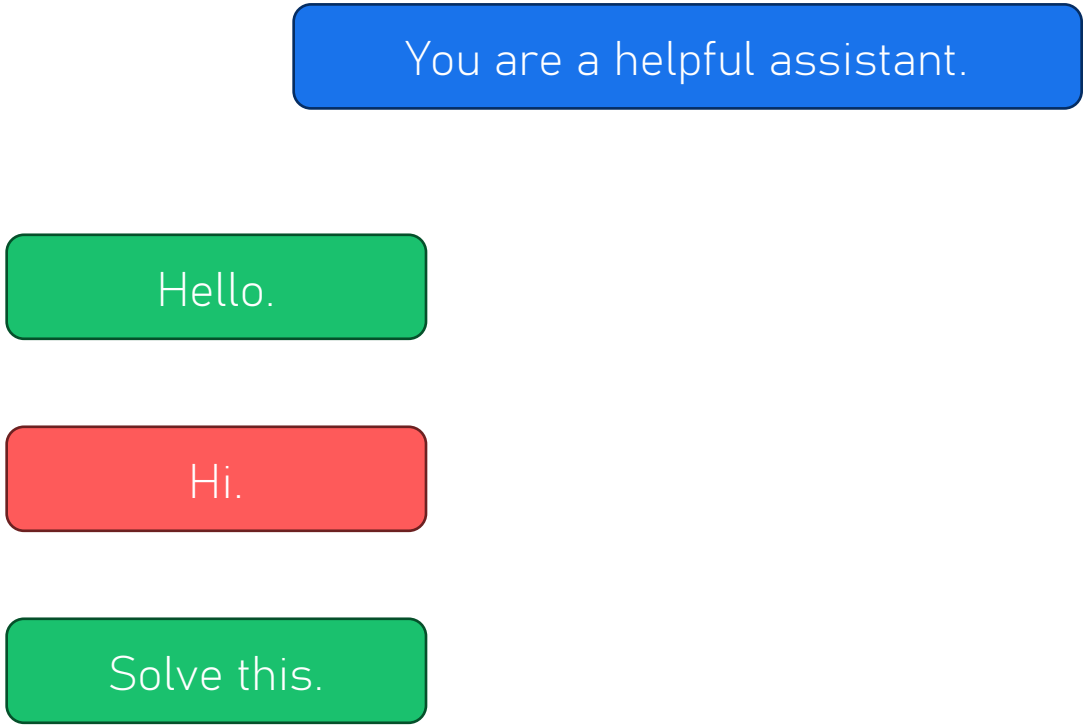
Duplicated Prefixes



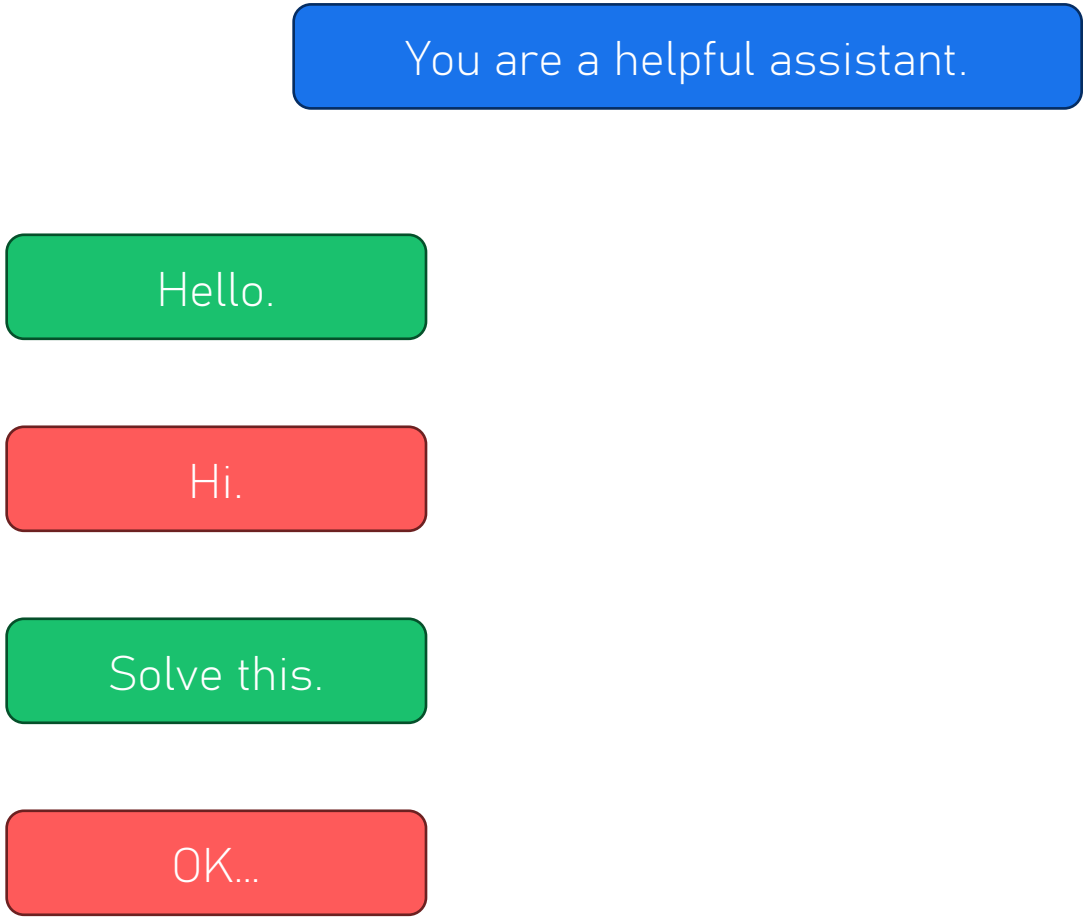
Duplicated Prefixes



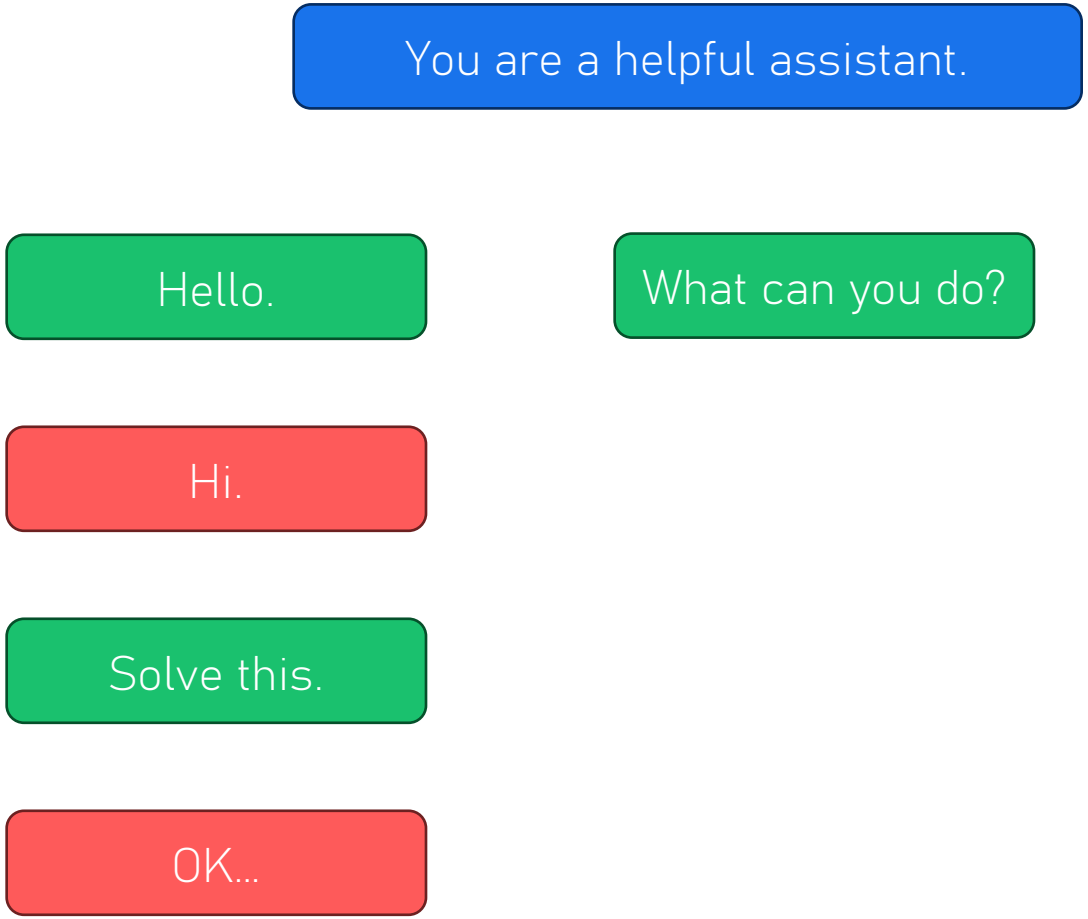
Duplicated Prefixes



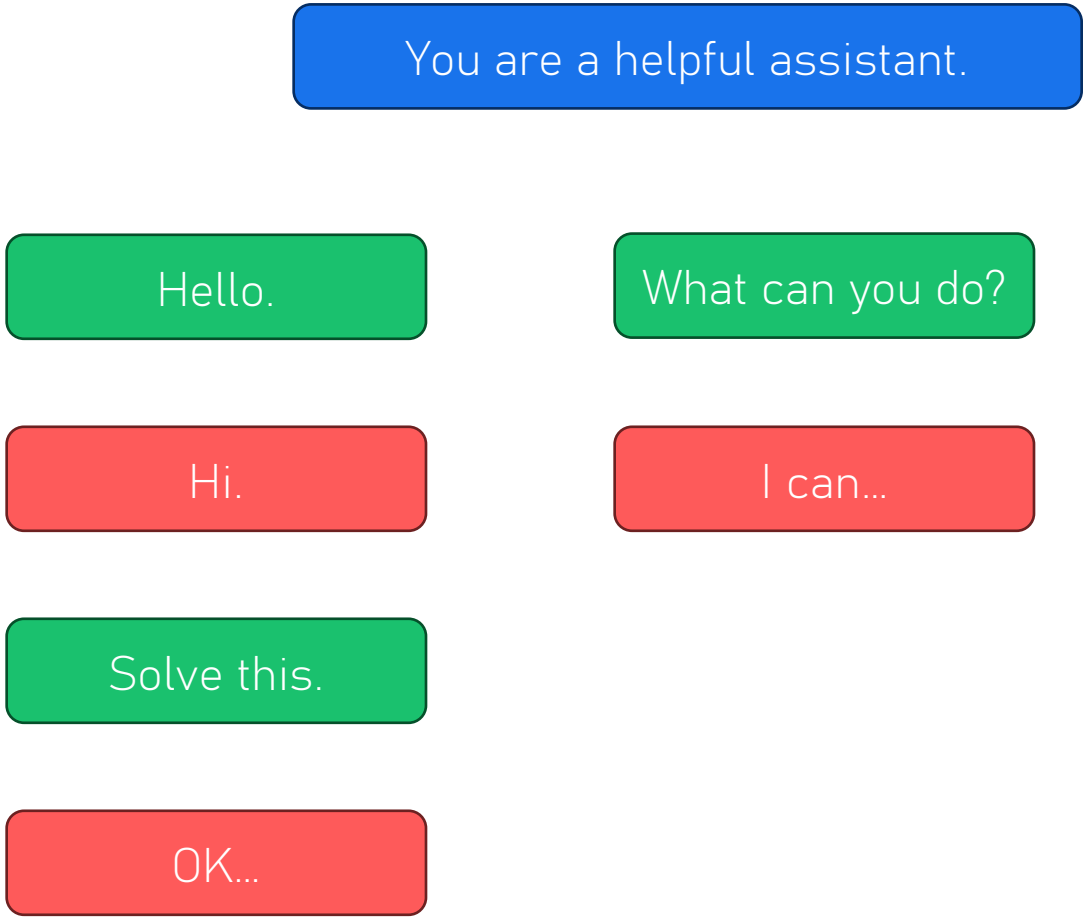
Duplicated Prefixes



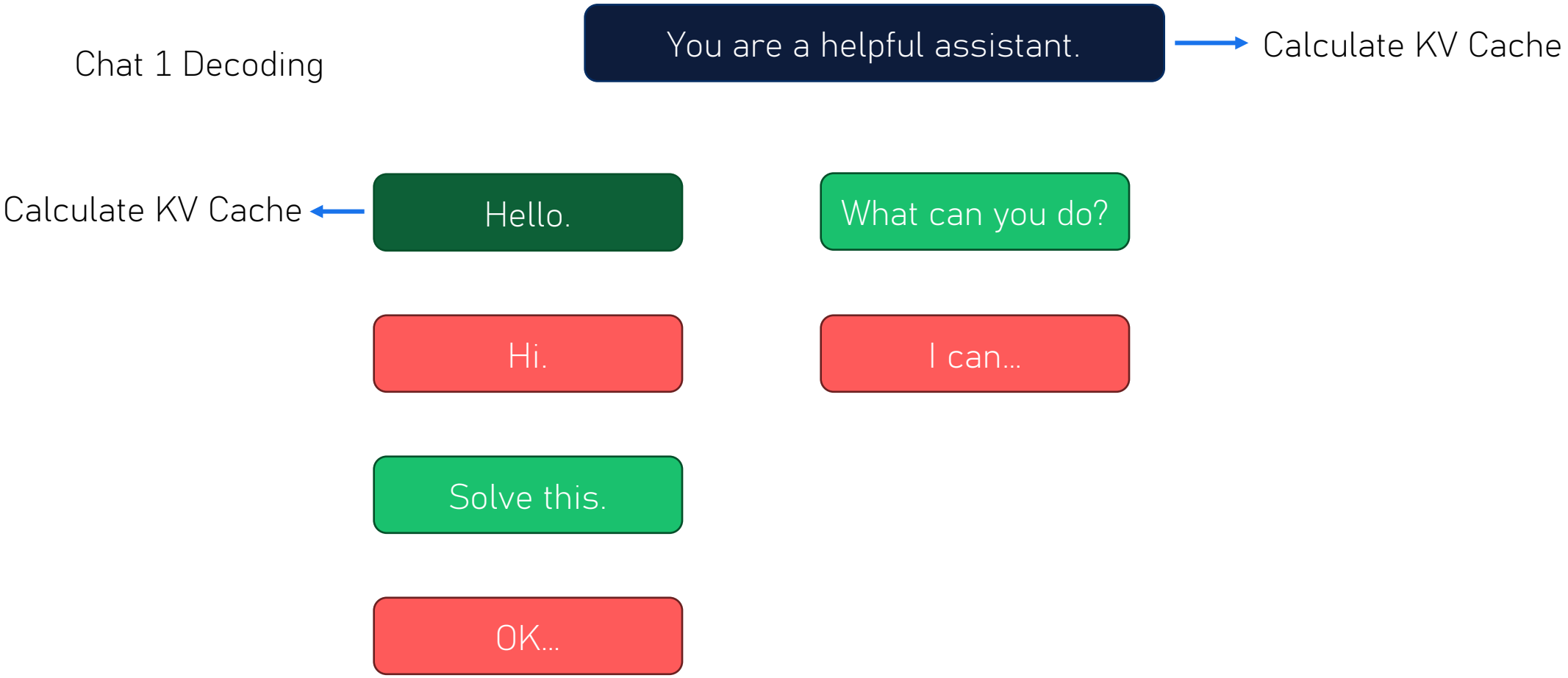
Duplicated Prefixes



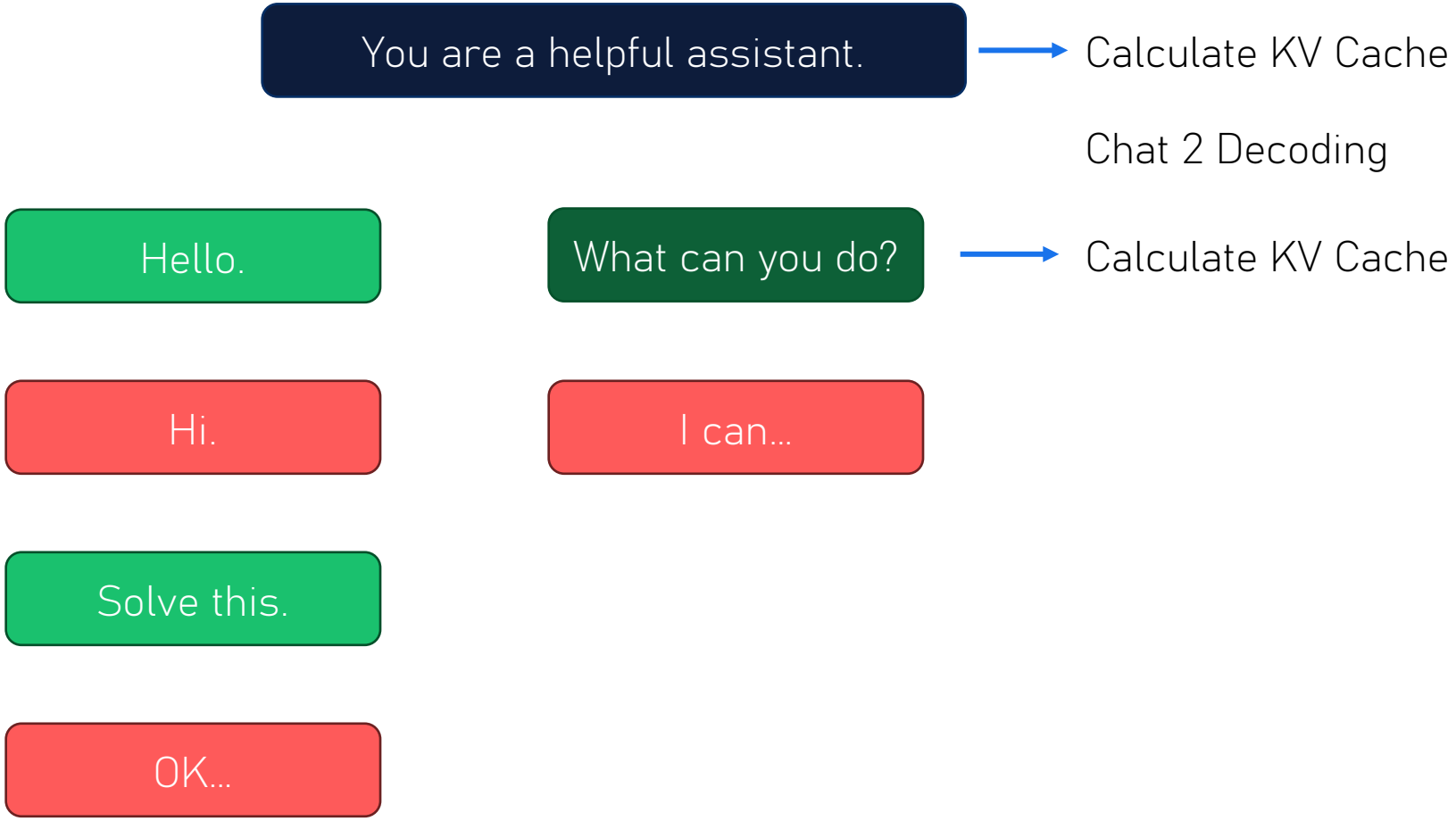
Duplicated Prefixes



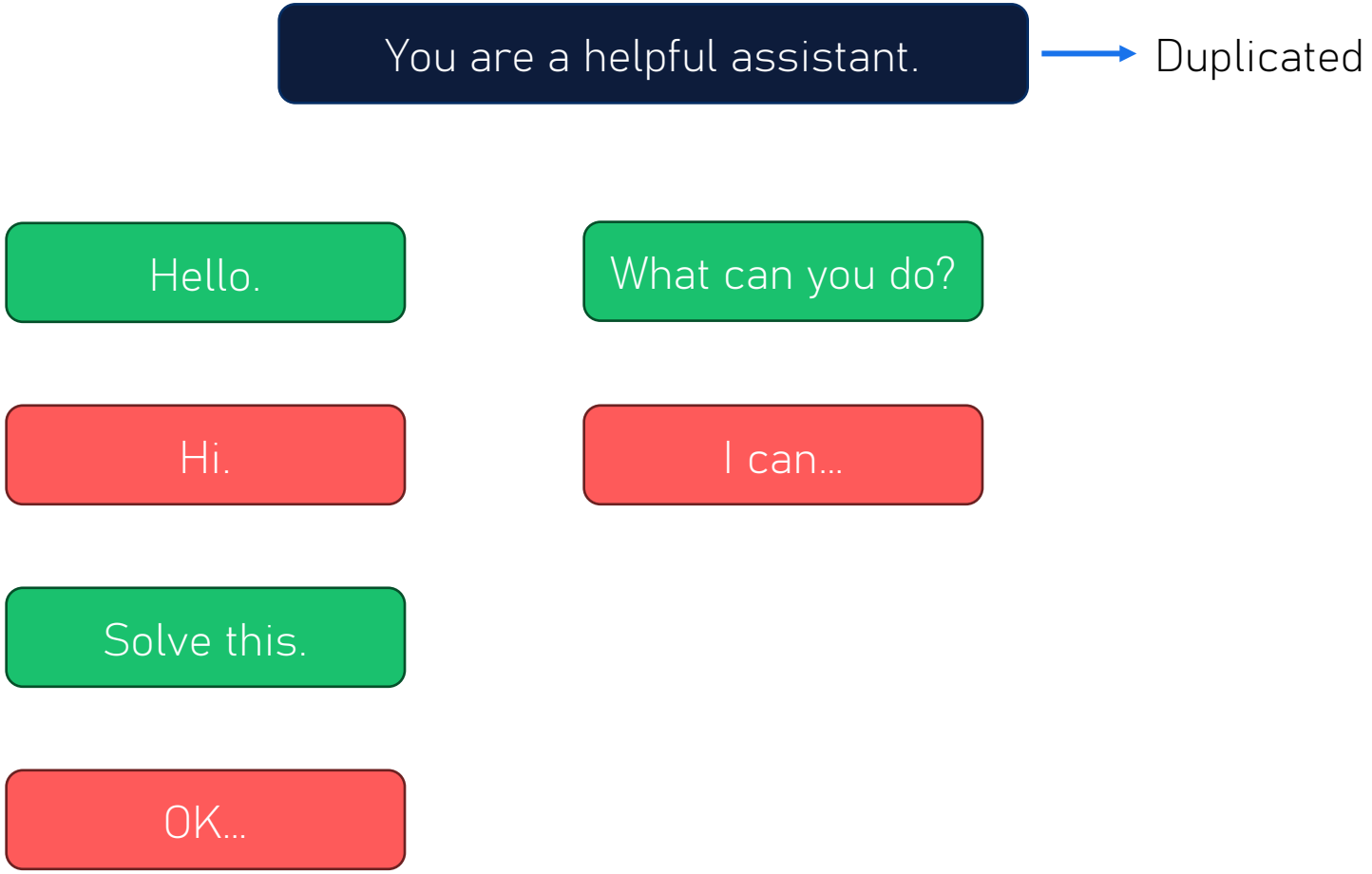
Duplicated Prefixes



Duplicated Prefixes



Duplicated Prefixes



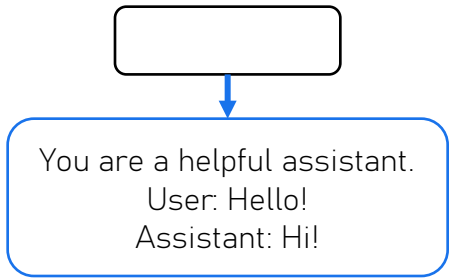
Structured Outputs

- The outputs should be in a specific format
  - E.g. JSON
- In the formats, sometimes there is only one possible next token
  - E.g. {“summary”:”…”}, the brackets, quotation marks, and colons are not replacable
  - So, computation for these tokens is redundant

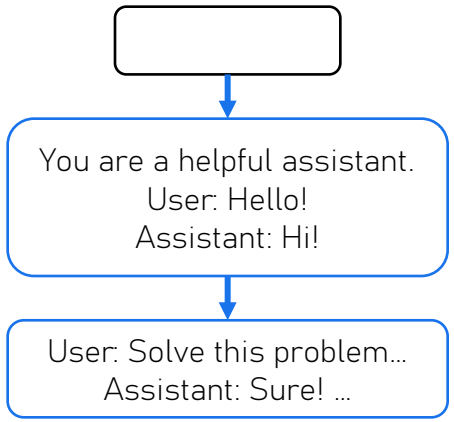
RadixAttention

- Use a radix tree to manage the mappings between tokens and their KV Cache
  - A radix tree is like a prefix tree, but it allows elements of varying lengths
- The KV Cache is managed in a page-like way, each page for one token
- Cache eviction policy
  - LRU (least recently used) leaves
  - No eviction currently being used ones (tracked by reference counters)

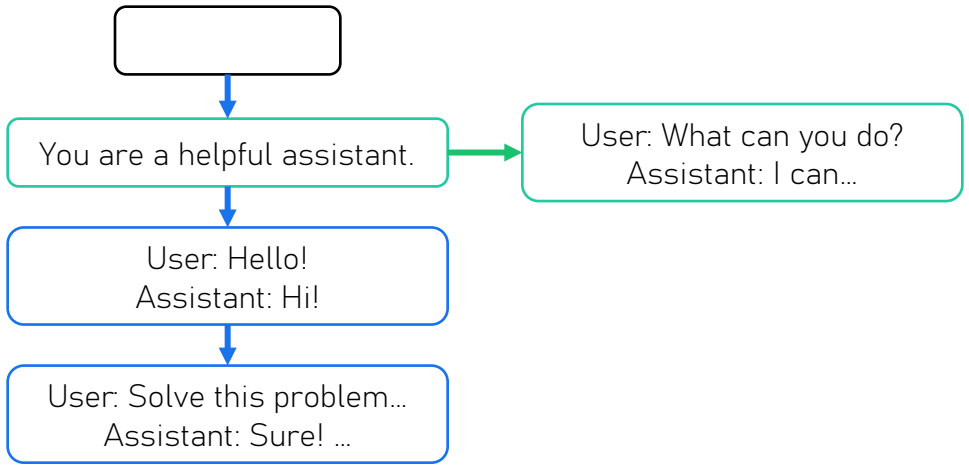
RadixAttention – Example



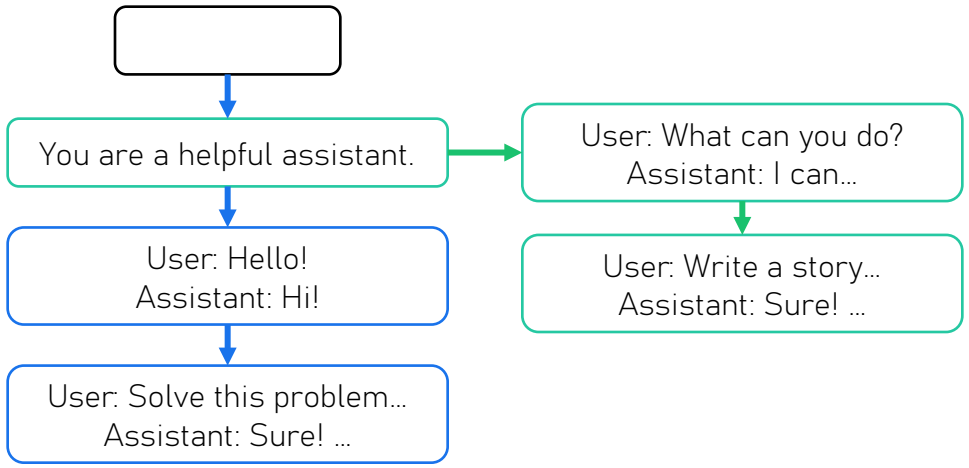
RadixAttention – Example



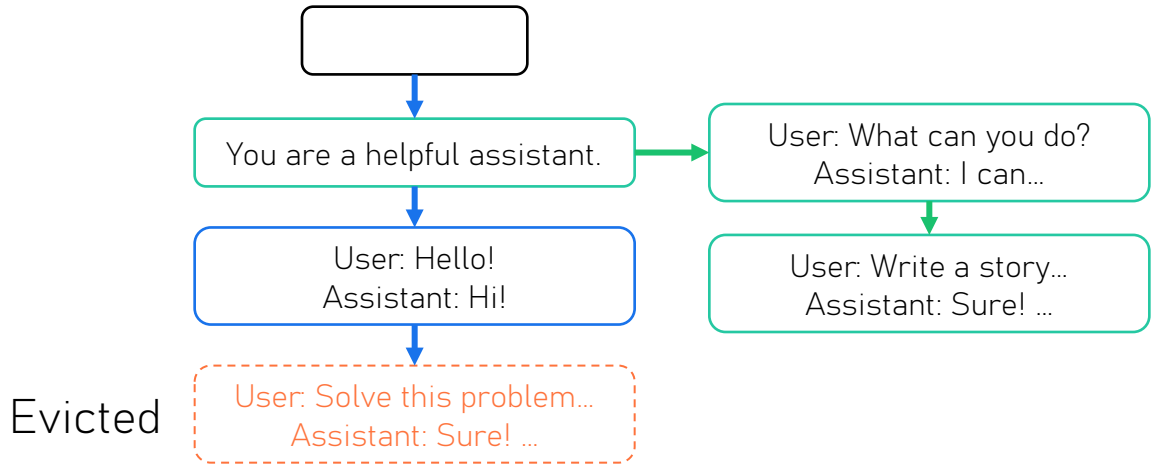
RadixAttention – Example



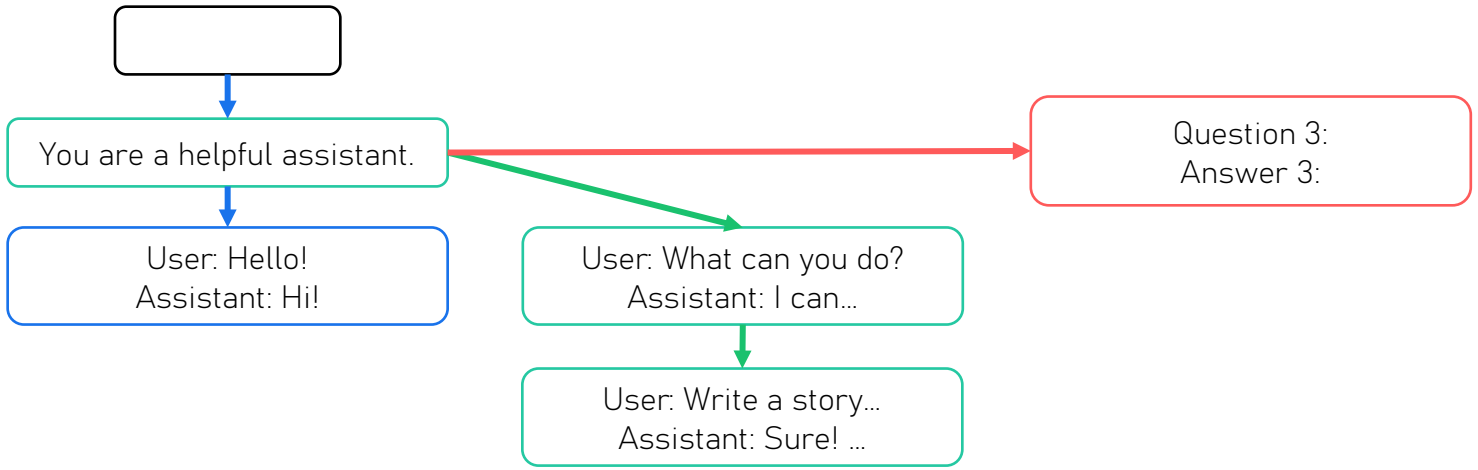
RadixAttention – Example



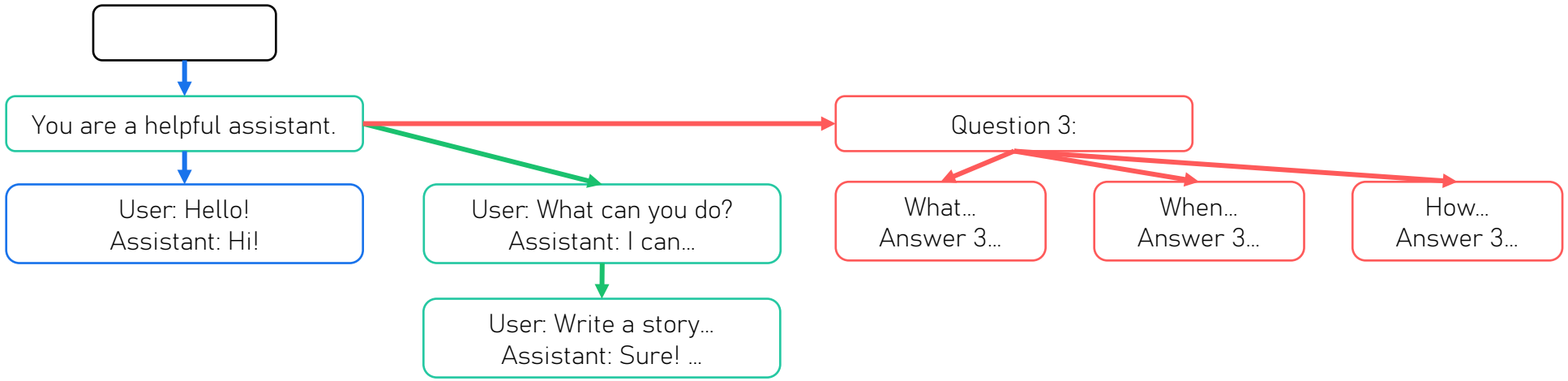
RadixAttention – Example



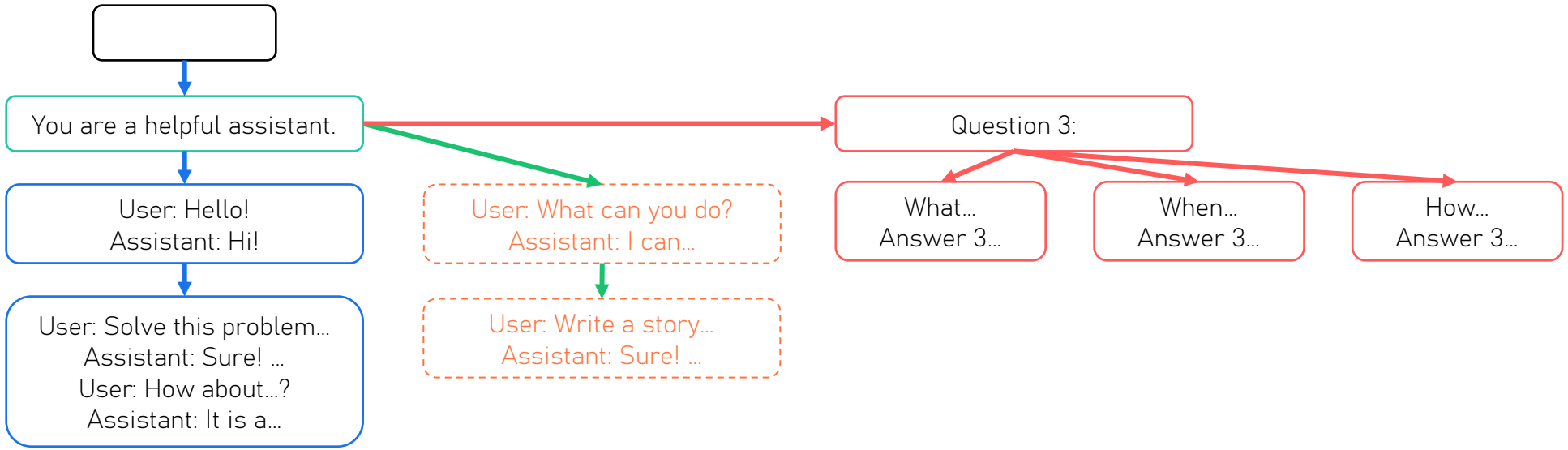
RadixAttention – Example



RadixAttention – Example



RadixAttention – Example

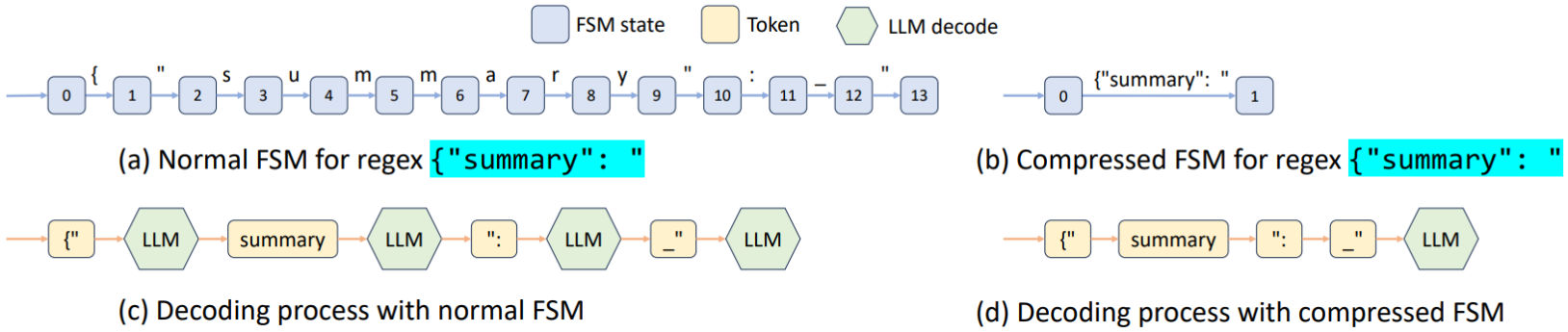


Cache-Aware Scheduling

- Cache hit rate = $\frac{number\ of\ cached\ prompt\ tokens}{number\ of\ prompt\ tokens}$
- To improve the cache hit rate, prioritize the request with longer matched prefixes
- May lead to starvation

Compressed Finite State Machine

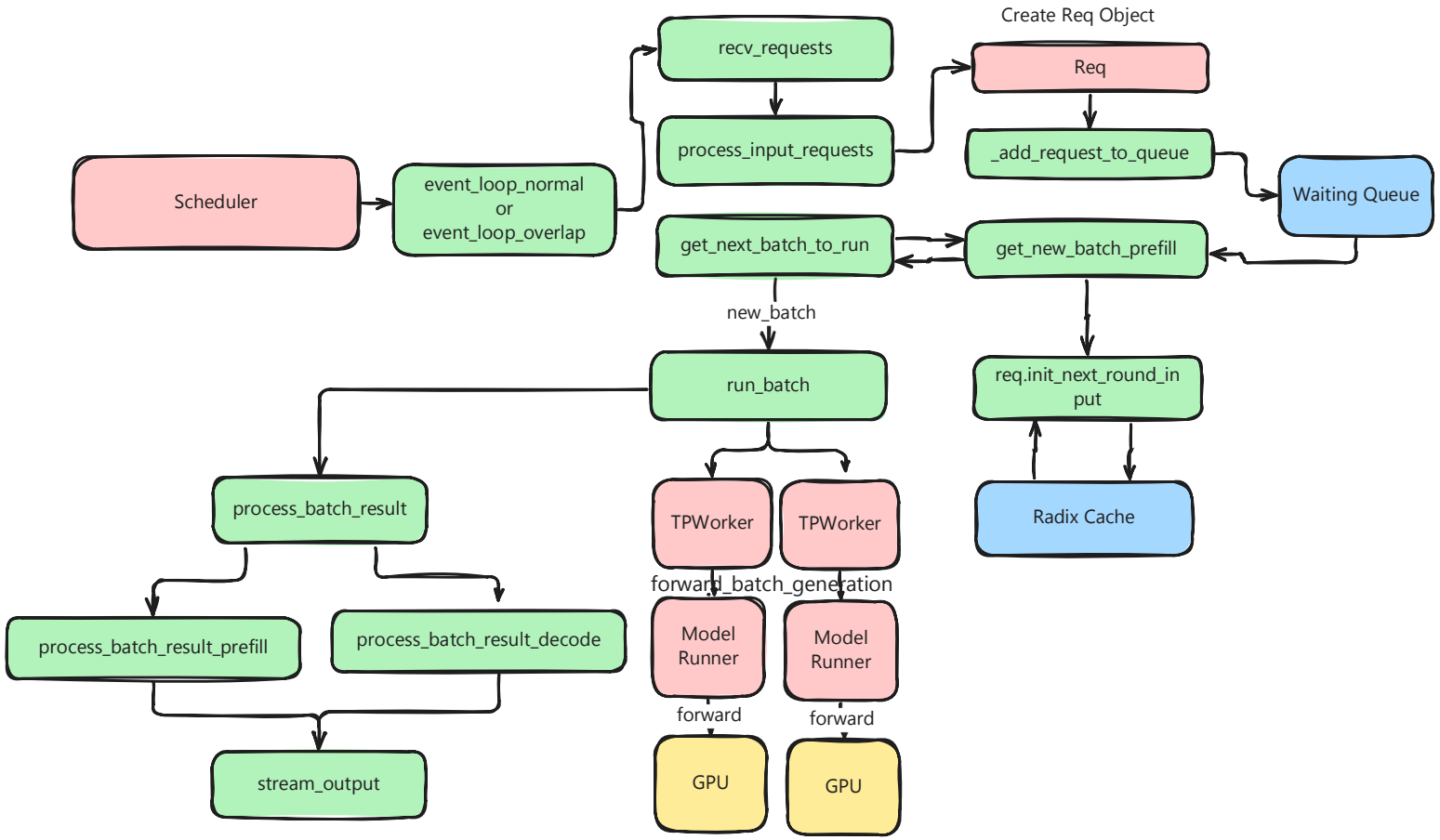
- Support defining output format constraints with regular expressions
- The regular expressions can be transformed to finite state machines
- Compress adjacent singular-transition edges into single edges



API Speculative Execution

- Speculate what to do next
- E.g.: s += context + "name:" + gen("name", stop="\n") + "job:" + gen("job", stop="\n")
  - Generate more tokens even when seeing the stop
- Enable reusing the additional generated outputs so it can save some API calls

SGLang Workflow



RadixAttention Implementation

python > sglang > srt > mem_cache > radix_cache.py > ...

```python

class TreeNode:

    counter = 0

    def __init__(self, id: Optional[int] = None):
        self.children = defaultdict(TreeNode)
        self.parent: TreeNode = None
        self.key: List[int] = None
        self.value: Optional[torch.Tensor] = None
        self.lock_ref = 0
        self.last_access_time = time.monotonic()

        self.hit_count = 0
        # indicating the node is loading KV cache from host
        self.loading = False
        # indicating the node is locked to protect from eviction
        # incremented when the node is referenced by a storage operation
        self.host_ref_counter = 0
        # store the host indices of KV cache
        self.host_value: Optional[torch.Tensor] = None
        # store hash values of each pages
        self.hash_value: Optional[List[str]] = None
        self.backuped_storage = False

        self.id = TreeNode.counter if id is None else id
        TreeNode.counter += 1
```

```python
class RadixCache(BasePrefixCache):
    def __init__(
        self,
        req_to_token_pool: ReqToTokenPool,
        token_to_kv_pool_allocator: BaseTokenToKVPoolAllocator,
        page_size: int,
        disable: bool = False,
        enable_kv_cache_events: bool = False,
    ):
        self.req_to_token_pool = req_to_token_pool
        self.token_to_kv_pool_allocator = token_to_kv_pool_allocator
        self.page_size = page_size
        self.disable = disable
        self.enable_kv_cache_events = enable_kv_cache_events
        self.kv_event_queue = []

        if self.token_to_kv_pool_allocator:
            self.device = self.token_to_kv_pool_allocator.device
        else:
            self.device = torch.device("cpu")

        if self.page_size == 1:
            self.key_match_fn = _key_match_page_size1
            self.get_child_key_fn = lambda key: key[0]
        else:
            self.key_match_fn = partial(_key_match_paged, page_size=page_size)
            self.get_child_key_fn = lambda key: tuple(key[:page_size])
        self.reset()

    ##### Public API #####
```

RadixAttention Implementation

```python
def _match_prefix_helper(self, node: TreeNode, key: List):
    node.last_access_time = time.monotonic()

    child_key = self.get_child_key_fn(key)

    value = []
    while len(key) > 0 and child_key in node.children.keys():
        child = node.children[child_key]
        child.last_access_time = time.monotonic()
        prefix_len = self.key_match_fn(child.key, key)
        if prefix_len < len(child.key):
            new_node = self._split_node(child.key, child, prefix_len)
            value.append(new_node.value)
            node = new_node
            break
        else:
            value.append(child.value)
            node = child
            key = key[prefix_len:]

            if len(key):
                child_key = self.get_child_key_fn(key)

    return value, node
```

```python
def evict(self, num_tokens: int):
    if self.disable:
        return

    leaves = self._collect_leaves()
    heapq.heapify(leaves)

    num_evicted = 0
    while num_evicted < num_tokens and len(leaves):
        x = heapq.heappop(leaves)

        if x == self.root_node:
            break
        if x.lock_ref > 0:
            continue

        self.token_to_kv_pool_allocator.free(x.value)
        num_evicted += len(x.value)
        self._delete_leaf(x)

        if len(x.parent.children) == 0:
            heapq.heappush(leaves, x.parent)

        self._record_remove_event(x)
```

RadixAttention Implementation

```python
class ForwardMode(IntEnum):
    # Extend a sequence. The KV cache of the beginning part of the sequence is already computed (e.g., system prompt).
    # It is also called "prefill" in common terminology.
    EXTEND = auto()
    # Decode one token.
    DECODE = auto()
    # Contains both EXTEND and DECODE when doing chunked prefill.
    MIXED = auto()
    # No sequence to forward. For data parallel attention, some workers will be IDLE if no sequence are allocated.
    IDLE = auto()
```

```python
def forward(
    self,
    q: torch.Tensor,
    k: torch.Tensor,
    v: torch.Tensor,
    layer: RadixAttention,
    forward_batch: ForwardBatch,
    save_kv_cache: bool = True,
    **kwargs,
):
    """Run forward on an attention layer."""
    if forward_batch.forward_mode.is_idle():
        return q.new_empty(q.shape[0], layer.tp_q_head_num * layer.v_head_dim)
    elif forward_batch.forward_mode.is_decode():
        return self.forward_decode(
            q,
            k,
            v,
            layer,
            forward_batch,
            save_kv_cache=save_kv_cache,
            **kwargs,
        )
    else:
        return self.forward_extend(
            q,
            k,
            v,
            layer,
            forward_batch,
            save_kv_cache=save_kv_cache,
            **kwargs,
        )
```

SGLang Usage

Evaluation Workloads

- MMLU, HellaSwag: few-shot examples
- ReAct Agents, Generative Agents: agent templates
- Tree-Of-Thought, Skeleton-Of-Thought: parallelize and reuse hints
- LLM Judges: parallelize different dimensions
- JSON Decoding: compressed finite state machines
- Multi-Turn Chats: reuse chat history
- RAG Pipelines: reuse context

Evaluation Workloads - MMLU

```python
def main(args):
    subjects = sorted(
        [
            f.split("_test.csv")[0]
            for f in os.listdir(os.path.join(args.data_dir, "test"))
            if "_test.csv" in f
        ]
    )

    # Build prompts
    arguments = []
    labels = []
    num_questions = []

    for subject in subjects[: args.nsub]:
        dev_df = pd.read_csv(
            os.path.join(args.data_dir, "dev", subject + "_dev.csv"), header=None
        )[: args.ntrain]
        test_df = pd.read_csv(
            os.path.join(args.data_dir, "test", subject + "_test.csv"), header=None
        )
        num_questions.append(test_df.shape[0])

        k = args.ntrain
        few_shot_examples = gen_prompt(dev_df, subject, k)
        while len(tokenizer.encode(few_shot_examples)) > 1536:
            k -= 1
            few_shot_examples = gen_prompt(dev_df, subject, k)

        for i in range(test_df.shape[0]):
            prompt_end = format_example(test_df, i, include_answer=False)

            arguments.append(
                {
                    "examples": few_shot_examples,
                    "question": prompt_end,
                }
            )

            label = test_df.iloc[i, test_df.shape[1] - 1]
            labels.append(label)
```

Evaluation Workloads - ReAct

```python
@sgl.function
def webthink(s, question, triplets):
    s += (
        """Solve a question answering task with interleaving Thought, Action, Observation steps. Thought can reason about the current situation, and Action can be three types: ···
        + question
    )
    for i in range(1, len(triplets) + 2): ···


def main(args):
    lines = read_jsonl(args.data_path)[: args.num_questions]
    arguments = [{"question": k, "triplets": v} for l in lines for k, v in l.items()]

    # Select backend
    backend = select_sglang_backend(args)
    sgl.set_default_backend(backend)

    states = []
    tic = time.perf_counter()
    states = webthink.run_batch(
        arguments,
        temperature=0,
        num_threads=args.parallel,
        progress_bar=True,
    )
    latency = time.perf_counter() - tic
```

Evaluation Workloads – Tree-Of-Thought

```python
def propose_plan(s, question, num_branches):
    s += sgl.user(
        """Please generate a high-level plan for solving the following question. As the first step, just say what method and idea you will use t
        + question
    )
    forks = s.fork(num_branches)
    forks += sgl.assistant(sgl.gen("plan", max_tokens=256, temperature=temp))
    return forks


def execute_plan(s, num_branches):
    s += sgl.user(
        """The plan looks good! Now, use real numbers and do the calculation. Please solve the question step-by-step according to the high-level
    )
    forks = s.fork(num_branches)
    forks += sgl.assistant(sgl.gen("answer", max_tokens=256, temperature=temp))
    return forks


def reflect_solution(s, num_branches):
    s += sgl.user(
        """Okay. Now, evaluate your own solution and give it a score on a scale of 1 to 5. Please do rigorous check of the correctness."""
    )
    forks = s.fork(num_branches)
    forks += sgl.assistant(sgl.gen("score", max_tokens=256, temperature=temp))
    return forks


def get_final_answer(s, num_branches):
    s += sgl.user(
        """Based on your reflection, do you change your mind? Now, give me the final answer after careful consideration."""
    )
    forks = s.fork(num_branches)
    forks += sgl.assistant(sgl.gen("final_answer", max_tokens=256, temperature=temp))
    return forks
```

Evaluation Workloads – LLM Judge

```
system_prompt = "Please serve as an impartial judge and rigorously evaluate the quality

dimension_prompts = [
    "Content: This refers to the essences of the essay. The substance should be well re
    "Organization and Structure: An essay needs to be properly structured with a clear
    "Argument and Analysis: The argument made in the essay should be logical, coherent
    "Clarity and Precision: The essay should be written in a clear and concise manner.
    "Grammar and Punctuation: Proper use of grammar and punctuation is vital in an acad
    "Referencing and Citation: An essay should contain proper citations and references
]


@sgl.function
def multi_dimension_judge(s, article):
    s += system_prompt
    s += "\n```\n" + article + "\n```\n\n"

    forks = s.fork(len(dimension_prompts))
    for i in range(len(dimension_prompts)):
        forks[i] += (
            "USER: Please judge the quality based on the following metric. "
            + dimension_prompts[i]
            + " Please provide a single-paragraph judgement. "
            + "Focus on the provided metric and do not say other things. "
            'End your judgement paragraph with the word "END"\nJUDGE:'
        )
        forks[i] += sgl.gen("judgement", max_tokens=256, stop="END")
    forks.join()

    s += "I will judge the quality based on the following metrics.\n"
    for i in range(len(dimension_prompts)):
        s += (
            dimension_prompts[i].split(":")[0]
            + ": "
            + forks[i]["judgement"].strip()
            + "\n"
        )
```

Homework

Read the paper of FlashInfer, and answer the following questions:

- 1. How does the BSR format unify the data structures? Explain and compare it with PageAttention of vLLM.
- 2. How does the load-balanced scheduling work?

Related links:

- Paper of FlashInfer: https://arxiv.org/pdf/2501.01005
- FlashInfer Git Repo: https://github.com/flashinfer-ai/flashinfer
- FlashInfer Website: https://flashinfer.ai/

Q & A