PREBLE: EFFICIENT DISTRIBUTED PROMPT SCHEDULING FOR LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Prompts to large language models (LLMs) have evolved beyond simple user questions. For LLMs to solve complex problems, today's practices are to include domain-specific instructions, illustration of tool usages, and/or long context such as textbook chapters in prompts. As such, many parts of prompts are repetitive across requests. Recent works propose to cache and reuse KV state of prompts. However, they are all confined to a single-GPU optimization, while production LLM serving systems are distributed by nature.

This paper proposes Preble, the first distributed LLM serving platform that targets and optimizes for prompt sharing. We designed a distributed scheduling system that co-optimizes KV state reuse and computation load-balancing with a new scheduling algorithm and a hierarchical scheduling mechanism. Our evaluation of Preble with real workloads and request arrival patterns on two open-source LLMs shows that Preble outperforms the SOTA serving systems by 1.5× to 14.5× on average latency and 2× to 10× on p99 latency.

1 Introduction

Recently, new capabilities and use cases of LLMs have created two common features not seen in traditional LLM usages. First, prompts to LLMs are significantly longer than generated sequences. For example, questions about a long document (19) or a video clip (45) are answered by LLMs with short answers. As another example, detailed instructions and illustrations for LLMs are vital in accomplishing complex tasks like solving advanced math problems (47). The latest LLMs like OpenAI o1 have built-in reasoning capabilities; even though the end users' prompts do not need to be long, these models internally add more context like chain-of-thought prompting (41), lengthening the total context length (30). As demonstrated by o1 and other model usage, oftentimes, the longer a prompt is, the better quality the model can generate. Long prompts with short generations imply that the prefill phase significantly outweighs the decoding phase. Thus, improving the prefill phase performance is crucial to the overall performance of LLM serving systems.

Second, prompts are partially shared across requests. For example, a long document or video is often queried many times with different questions (19); different requests using the same tools share tool instructions in tool-augmented LLMs (11); chain- or tree-structured prompting calls an LLM in steps, with each subsequent step reusing context from previous steps (47; 50; 46). Real-world production systems like Anthropic have also reported the wide existence of long and shared prompts (2).

Recent works (52; 9; 26) propose to cache computed key-value (KV) state in GPU memory and reuse the cached KV when a new request sharing a prompt prefix arrives. These works aim to improve the serving performance of LLMs with long and shared prompts in a *single GPU* setting. However, in-production LLM serving systems typically utilize a distributed set of GPUs to serve user requests. Current distributed LLM serving systems are not prompt-cache-aware; they attempt to distribute LLM computation load equally across GPUs to achieve high cluster-level GPU utilization. Yet, this distribution could result in requests with shared prefixes being sent to different GPUs, causing KV computation at all these GPUs that could otherwise be avoided if prefixes are cached and reused on the same GPU. On the other hand, a naive solution that always sends requests with shared prefixes to the same GPU would result in imbalanced loads and low overall GPU utilization because the GPU that initially serves a request with a popular prefix will accumulate a huge load of new requests all trying to reuse the calculated prefix KV.

To properly design a distributed LLM serving system for long and shared prompts, we first perform a comprehensive study of five typical LLM workloads: LLM with tool calling (10), LLM as embodied agents in virtual environments (14), LLM for code generation (28), embedded video QA (45), and long document QA (19). We find their prompts to be 37× to 2494× longer than generated sequences, and 85% to 97% tokens in a prompt are shared with other prompts. Additionally, a request often shares prompts with multiple other requests at different amounts (*e.g.*, one prefix sharing being the initial system prompt, one sharing being the system prompt plus tool A's demonstration, and one being the system prompt plus tool A and tool B's demonstrations). We also analyze the Azure LLM request trace (32) and found these end-user traces to have large prompt-to-output ratios as well. Additionally, this trace shows that requests arrive at varying speeds over time and across LLM usages, making designing a distributed LLM serving system challenging.

Based on our findings, we propose a distributed LLM request scheduling algorithm called *E2* (standing for *Exploitation* + *Exploration*) that co-designs model computation load-balancing and prefix-cache sharing. E2 allows requests to *exploit* (*i.e.*, reuse) computed prompt prefixes on the same GPU but also gives chances for requests with shared prefixes to *explore* other GPUs. E2 chooses exploitation when the amount of recomputation saved is larger than that of new computation, which happens when the number of shared prefix tokens is larger than the remaining non-shared tokens. Otherwise, E2 chooses exploration. For exploitation, we send the request to the GPU that caches the longest-matched prefix.

When E2 decides to explore GPUs, it chooses the GPU with the lightest "load", using a *prompt-aware load* definition we propose. This prompt-aware load includes three parts all calculated as GPU computation time. The first part is a GPU's computation load in a recent time window H, which is measured by the total prefill time and decode time incurred by all the requests in H. The second part is the cost of evicting existing KVs on the GPU to make memory space to run the new request. The third part is the cost of running the new request on the GPU. When calculating these three parts, we account for prompt sharing that have or would occur on the GPU by treating the reusable computation as zero cost. E2 picks the GPU with the lowest sum of the three parts to explore, which balances loads while accounting for cached prompt behavior.

Centered around the E2 scheduling algorithm, we build *Preble*, a distributed LLM serving system that aims to provide high serving throughput and low request average and tail latency for long and shared prompts. Preble consists of a global, request-level scheduler and a per-GPU, iteration-level scheduler. Apart from E2, Preble incorporates several novel designs to tackle practical LLM challenges. First, with the basic E2 algorithm, a prefix is cached at a GPU after its initial assignment until its eviction. However, the amount of requests sharing it can change over time, which can cause load imbalance. To mitigate this issue, Preble detects load changes and redirects requests from a heavily loaded GPU to a light GPU. If the load hitting a popular prefix increases beyond what a single GPU can handle, Preble automatically scales (autoscales) the prefix by replicating it on multiple GPUs.

Second, prefill and decoding phases have different computation needs, as discovered and tackled by a set of recent works (5; 53; 32; 4). We propose a new way to solve this problem based on our insight that a prompt hitting a cached prefix can be treated as decoding-phase computation, while a missed prompt can be treated as prefill-phase computation because of the high prompt-to-decoding token length ratio. Thus, our global scheduler tries to mix the two types of requests on a GPU to balance prefill and decoding computation needs.

Finally, unlike existing works that either honor request fairness or maximize prefix matching, Preble aims to achieve high prefix reusing while ensuring fairness, which is important in multi-tenancy environments. We achieve this by assigning priorities to waiting requests based on their prefix cache hit ratio and giving each priority their respective quota of requests to serve.

We implement Preble as a standalone layer on top of slightly modified vLLM (18) and SGLang (52), two popular open-source LLM serving systems both supporting single-GPU prefix caching. We evaluate Preble using our studied five workloads and the Azure request arrival pattern with the Mistral 7B model (16) and the Llama-3 70B model (24) on a four-Nvidia-A6000 GPU cluster and an eight-Nvidia-H100 GPU server. Our results show that Preble outperforms SGLang by 1.5×-14.5× and 2×-10× on average and p99 average request latency and similarly when compared to vLLM.

Overall, this paper makes the following key contributions: **(1)** The first study of LLM workloads with long and shared prompts, resulting in four key insights. **(2)** E2, a new LLM request scheduling algorithm with the idea of exploitation and exploration integration. **(3)** Preble, the first distributed LLM serving system that targets long and shared prompts. **(4)** A comprehensive evaluation of Preble and SOTA LLM serving systems on two popular open-source LLMs, five real workloads, and two GPU clusters. Preble is publicly available at *[link anonymized]*.

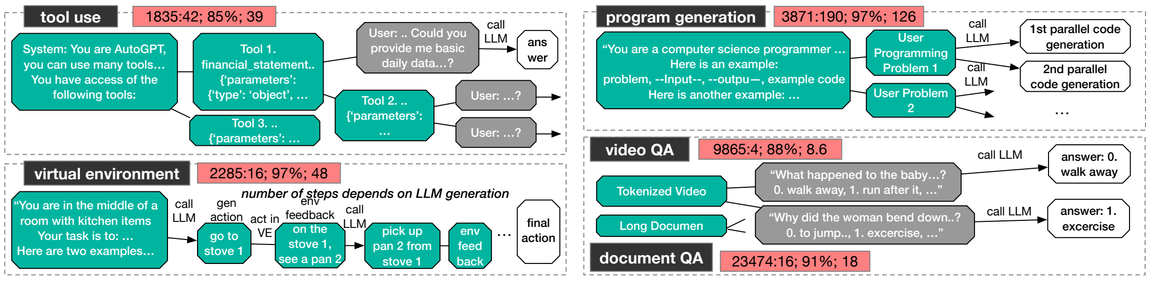
tool use
1835:42; 85%; 39
System: You are AutoGPT, you can use many tools… You have access of the following tools:
Tool 1. financial_statement.. {'parameters': {'type': 'object', …
Tool 3. .. {'parameters': …
Tool 2. .. {'parameters': …
User: .. Could you provide me basic daily data…?
call LLM
ans wer
User: …?
User: …?
virtual environment
2285:16; 97%; 48
number of steps depends on LLM generation
"You are in the middle of a room with kitchen items Your task is to: … Here are two examples…
call LLM
gen action
go to stove 1
act in VE
env feedback
on the stove 1, see a pan 2
call LLM
pick up pan 2 from stove 1
env feed back
…
final action
program generation
3871:190; 97%; 126
"You are a computer science programmer … Here is an example: problem, --Input--, --outpu—, example code Here is another example: …
User Programming Problem 1
User Problem 2
call LLM
call LLM
1st parallel code generation
2nd parallel code generation
…
video QA
9865:4; 88%; 8.6
Tokenized Video
Long Documen
"What happened to the baby…? 0. walk away, 1. run after it, …"
"Why did the woman bend down..? 0. to jump.., 1. excercise, …"
call LLM
call LLM
answer: 0. walk away
answer: 1. excercise
document QA
23474:16; 91%; 18

Figure 1: **Prompt Sharing Features of Five Workloads.** *Green boxes represent shared prefixes. Grey boxes are non-shared prompts. White boxes are output generation. Red boxes contain statistics in average values: "prompt-length:output-length; shared token percentage; number of requests sharing a sequence".*

2 Real-World Long and Shared Prompts

This section briefly presents our study results of five LLM use cases and an end-user LLM use trace: tool use (35), embodied agent in a virtual environment (11; 14), software program generation (28), video QA (45), long document QA (19), and the Azure LLM usage trace (32) (which contains chat and code-generation usages). We pick the five workloads to study and evaluate, as they resemble long-and-shared-prompt use cases reported in production (2). Figure 8 demonstrates the prompt usages and overall features of these workloads. Appendix A presents our full study methodology and results.

Overall, our study shows that prompts are significantly longer than output lengths in the five workloads and the Azure trace, ranging from 4× (Azure chat) to 2494× (video QA). Moreover, prompt sharing ranges from 85% (*i.e.*, 85% prefix tokens in a prompt are shared with at least one more request) to 97% across the five workloads, with embodied agents and program generation having the highest sharing amount. Additionally, a common sequence in a workload is shared by 8.6 to 126 requests on average, with different deviations across workloads. Finally, the Azure trace indicates high variations in total request loads manifested as different end-user request inter-arrival times that range from 2 microseconds to 217 seconds.

Our study results indicate that LLM serving systems for such workloads should focus on optimizing prefill computation, and a viable way is to cache and share prefixes. However, the variations in prompt-sharing features and request arrival patterns make it challenging to build an efficient distributed serving system.

3 Preble Design

We now present the E2 algorithm and the design of Preble, beginning with the overall system architecture of Preble, followed by its global scheduler and local scheduler designs. Preble significantly improves serving speed and reduces request latency for workloads with long and shared prompts. For workloads without any shared prompts, its behavior and performance are the same as SOTA LLM serving systems like vLLM (18).

3.1 Overall System Architecture

Preble is a distributed GPU-based LLM serving system supporting both data parallelism and model parallelism. While its model parallelism support is standard (*e.g.*, tensor parallelism), Preble's scheduling of requests on data-parallel GPUs is designed specifically for long and shared prompts.

We propose a two-level scheduling system where a global scheduler performs *request-level* scheduling decisions and orchestrates the overall load balancing across GPUs, while a per-model-instance local scheduler performs *iteration-level* scheduling for requests assigned to the GPU, as shown in Figure 2. Depending on the GPU cluster topology, the global scheduler may be deployed on a separate server for a multi-server GPU cluster or on the same server as a single-server-multi-GPU cluster. The local scheduler manages one model instance (multiple GPUs when using model parallelism, single GPU when not) and runs on the CPU of the same server as the GPUs. Our current implementation of Preble scales to at least 70 to 391 GPUs. To offer larger, data-center-level scales, one can deploy several Preble clusters, each having one global scheduler.

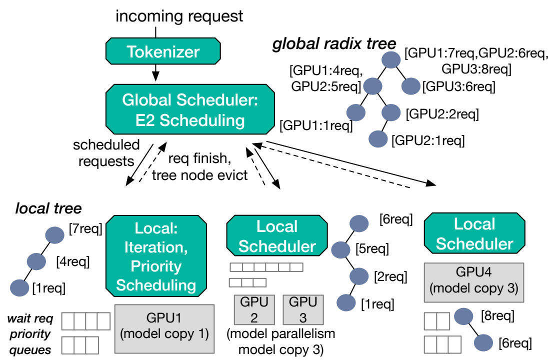

Figure 2: **Preble Architecture.**

This design offers several benefits: 1) by having all requests in a cluster go through the global scheduler, we have a centralized place to maintain a global view of cluster load and prompt caching information, both being essential for E2; 2) by performing coarse-grained, request-level scheduling, a single global scheduler can scale to hundreds of GPUs, avoiding the complexity of maintaining multiple distributed global schedulers for a cluster; and 3) by performing fine-grained, iteration-level scheduling at each GPU, the local scheduler can quickly adapt to GPU resource and request availability changes and make precise decisions.

3.2 E2 Global Scheduler

We now present our global scheduler design, which centers around the E2 distributed scheduling algorithm.

Global scheduler data structures. The global scheduler maintains several data structures to assist its prompt-aware request scheduling. The primary data structure is *global prefix trees*, implemented as radix trees (42). Each tree has a distinct root storing the shared prefix of all prompts under the tree. When inserting a new request to the tree, we match its tokens from the beginning (*i.e.*, prefix matching) until no match exists, and we insert the remaining tokens as a new leaf node. If no match exists at all, we create a new tree with this request's prompt as the root node. If an existing tree node only matches partially to the new request (*i.e.*, the prefix of a node matches a sub-sequence of the new request), we split the node into the matched part and the remaining part. For each tree node, we record three pieces of information: the number of tokens in the tree node, the set of GPUs caching the tree node KVs, and the per-GPU number of requests sharing the tree node in a history window H. When a tree node has no caching GPU and there is no request within the window H in the whole system sharing it, we remove it from the tree.

Per-request scheduling policy. To schedule a request, the global scheduler uses our proposed E2 scheduling algorithm, as illustrated in Algorithm 1. It first matches the request's prompt in the global prefix trees. When the amount of recomputation saved (number of tokens in the matched prefix) is larger than the amount of new computation (number of tokens in the remaining prompt), we favor exploitation over exploration because the gain of saved GPU resources (computation for matched tokens) is higher than the load (computation for unique tokens) that can potentially be balanced in the GPU cluster. For such requests, E2 *exploits* existing cache by assigning the request to the GPU that caches the tree node with the longest tokens in the matched prefix. If multiple such GPUs exist, E2 chooses the GPU with the lightest request load using the load calculation to be introduced next.

If the matched prefix is shorter than the remaining tokens, E2 *explores* the best GPU to run the request based on our proposed new prompt-aware "load cost" definition. Exploration gives E2 a chance to distribute load to different GPUs, which is the key to striking long-term cluster execution efficiency. E2 unifies three types of costs when calculating the per-GPU load: the computation cost aggregated across all requests within a time window, the recomputation cost needed for evicting memory to run the new request, and the computation cost of the new request. E2 calculates all three costs as GPU computation time and finds the GPU with the lowest sum. Instead of profiling the actual computation time, we only maintain token counts at the global scheduler, which largely reduces the system overhead. We leverage transformer-based LLMs' properties that the computation amount of prefill and decoding are proportional to the number of prompt tokens and generated tokens (Figures 9 and 10). Below and in Algorithm 2, we detail the three calculations for scheduling a new request R_k.

The first cost is the *overall* GPU computation load L_i for GPU_i. We capture a recent load history on GPU_i with a time window H with a default value of 3 minutes (we test different H lengths and find the results insensitive to it). We do not use GPU_i's load at the exact request scheduling time for two reasons: 1) a GPU's load can change between the time of scheduling R_k to the time of running it, and 2) the placement of a prefix has a longer-term effect than a single load in time because of other requests' future exploitation of it. For each request R_r in the history, we estimate its prefill time PT_r with a regression function using the number of tokens in R_r that do not match any prefixes on GPU_i; we estimate its decoding time DT_r with another regression function using the average request output length observed on GPU_i in window H. We have the first category of load, $L_i = \sum_{r \in W}(PT_r + DT_r)$, where W is the set of requests in the window H. The regression functions used in this calculation are captured from offline profiling for each GPU type. Note that even though the number of output tokens is not known a priori, our workload study (Appendix A) shows that it is small and similar across a type of workload. Thus, we use the average output length in H as the estimated decoding length for DT_r.

The second cost is the potential cost to free GPU memory so that the new request, R_k, can run. Given that GPUs run at full capacity with our and existing serving policies (5), we expect this cost to always occur. Intuitively, the more tokens in a sequence that are shared and by more requests, the more costly it is to evict the sequence. Thus, to calculate the eviction cost, M_i, the global scheduler first uses the eviction algorithm to be discussed in Section 3.3 to find the tree nodes on GPU_i that would be evicted to run R_k. For each such tree node j, its eviction cost is the recomputation time of the evicted tokens multiplied by the hit rate of the node, N_j. Thus, we have $M_i = \sum_{j \in E} PT_j \times N_j$ where E is the eviction node set, PT_j is the prefill time for the length of tree node j, and N_j is the the number of requests sharing tree node j in history H over the total number of requests on GPU_i in H. Note that we do not include the decoding time, as a request's decoding time is unaffected by prefix cache eviction, and decoding costs have already been counted in L_i.

The third cost is the actual cost, P_i, to run the new request R_k on GPU_i, which is simply the prefill time of the missed tokens in request R_k. We do not count its decoding time, as it is the same across GPUs, and our goal is to compare the per-GPU load across GPUs.

The total cost of assigning the current request to GPU_i is $L_i + M_i + P_i$ and we choose the GPU with the lowest total cost to assign the request to.

Post-assignment load adjustment. With the above algorithm, after the global scheduler assigns a request to a GPU, its prefix lives there until its eviction. This greedy approach works well in cases where the load to a prefix is relatively stable but not otherwise. We propose two ways of managing post-assignment load changes. The first way shifts load between GPUs and is applicable when the load surge can be handled by a single GPU in the cluster. The global scheduler maintains a per-GPU load as discussed above. If the most heavily loaded GPU's load is more than Th_{bal} times higher than the lightest GPU, it shifts load from the former to the latter until their difference is below Th_{bal}. Th_{bal} is configurable and can be deducted from profiling GPU and LLM types. To perform this load rebalancing, we direct future requests that are supposed to exploit the heavy GPU to the light GPU.

The second way is to auto-scale a prefix by replicating it and splitting its subtree by load when we detect that a certain prefix's request load is still too high (average queueing time doubles over H) even after the above load rebalancing. We calculate the subtree's load using Algorithm 2.

Prefill-decoding balancing. From our study results in Appendix A and reported by others, LLM prefill has a larger compute-to-memory ratio than decoding, causing inefficient GPU resource utilization and performance degradation. While various recent works tackle the prefill-decoding imbalance problem by chunking prefills (5) and prefill-decoding disaggregation (53; 32; 37; 13; 33), we propose a new way of solving the problem leveraging prompt sharing features at a cluster level. Our insight is that a request with its entire prompt shared and cached would only perform the decoding phase. Thus, it can be treated as a decoding-phase computing unit. Meanwhile, a request with a long prompt not cached and a short output length can be treated as a prefill-phase computing unit. A partially cached prompt can be treated as being between the prefill- and decoding-phase units. Thus, we can balance prefill-decoding by combining requests with more or less prompt sharing instead of or in addition to existing balancing techniques.

Specifically, when a request is about to be explored, the global scheduler first considers the prefill and decoding balancing for each GPU. If a GPU is heavily loaded with decoding-phase computing units, the global scheduler directs the current request to it, as a request to be explored will incur recomputation for prompt and is considered a prefill-phase unit. We prioritize this policy over the load-cost comparison (Algorithm 2) because a GPU with heavy decoding has unused computation capacity that we can almost freely use. The global scheduler performs the load-cost comparison if all GPUs have relatively balanced decoding-prefill loads. Apart from this prefill-decoding balancing performed at the global scheduler, our local scheduler also performs traditional chunked prefill for each GPU (Section 3.3).

3.3 Local Scheduler

Local scheduler mechanism. The local scheduler schedules the requests that the global scheduler assigns to its managed GPU(s). Similar to existing LLM serving systems (48; 18; 52; 6; 25; 38), we run one local scheduler per GPU and schedule requests at the iteration level. Each local scheduler maintains a request wait queue, a prefix (radix) tree, and the number of active requests sharing each prefix tree node.

When a new request arrives, the local scheduler matches it to the local prefix tree and updates the tree accordingly. It also inserts the request into the waiting queue. After each model iteration, the local scheduler forms the next batch by selecting waiting requests using a priority-based algorithm to be discussed next. If a selected request has a long and non-shared prompt, we chunk the prompt similar to Sarathi (5). If the GPU memory is not enough to run the batch, the local scheduler selects a tree node(s) or part of a tree node (if a part is enough) to evict based on the request accessing time (LRU) of tree nodes. The local scheduler then asynchronously informs the global scheduler about the eviction, and the latter processes it in the background.

Waiting queue request ordering. Today's LLM serving systems schedule requests in the waiting queue according to FCFS (18) or prefix sharing (52) (serve the request with the highest sharing amount the first). The former ignores prompt sharing and results in more recomputation; the latter ignores fairness and could result in starvation (43). We propose a priority-based wait queue scheduling policy that considers both prefix sharing and fairness. Specifically, we create P (a configurable parameter) priority groups and assign a request to the priority group according to its cached token percentage. For example, if 63 out of 100 tokens in a request's prompt are cached on the GPU and P is 10, it will be assigned priority six. When selecting requests to form the next batch, the scheduler proportionally selects requests from each priority group, with the higher priority group getting more requests selected than lower priority ones. For example, if 55 requests are to be selected to form a batch, the scheduler picks ten from priority group 10, nine from priority 9, etc.

4 Implementation and Evaluation Results

4.1 Implementation

We implemented Preble as a standalone layer to perform distributed LLM serving. As such, Preble can be added to any existing serving systems with no or minimal changes — we currently support vLLM (18) and SGLang (52) as two backends.

Global scheduler scalability. In implementing the global scheduler, we use a few techniques to improve its scalability. Incoming requests are first tokenized by a parallel tokenization layer. Afterward, the global scheduler spawns asynchronous request handlers to schedule requests. Access to the global radix tree during request handling is lock-free, as most operations are read-only. The only exceptions are updating a GPU to be assigned to a tree node and the increment of request count hitting the tree node, both of which can be expressed as atomic instructions. Additionally, the global scheduler maintains a current load count for each GPU by updating it every time a new request is assigned to it or when it evicts a tree node. Thus, our realization of the E2 algorithm is performance-efficient. Finally, to ensure foreground request performance, the global scheduler runs non-request-scheduling tasks such as rebalancing and eviction bookkeeping in the background with separate threads.

4.2 Workloads and Environments

Workloads. We evaluate our results on five LLM use cases: LLM generation with tool demonstration and calling (35), LLM interacting with virtual environments as an embodied agent (11; 14), LLM for software program generation (28), answering questions about videos (45), and answer questions about long documents (19). Their properties are presented in Appendix A.

For each workload, we sample enough requests to fulfill the request-per-second (RPS) needs and GPU setup (*e.g.*, a larger GPU or more GPUs can handle more). For experiments other than the ones using the Azure Inference Trace, we set the inter-arrival time using a Poisson distribution with a mean that corresponds to the RPS we test (X-axis in most figures). We then run the experiments until stable state is reached and lasts for a significant length.

LLMs and environments. We test Preble and baselines using two popular open-source Large Language Models, the Mistral 7B model (16) and the Llama-3 70B model (24). We run our experiments in one of the two local testbed environments: a two-server cluster each containing two NVidia A600 GPUs and one eight NVidia-H100-GPU server.

Baseline. Our baselines are serving systems that support single-GPU prefix sharing, including SGLang (52) and vLLM (which recently added a beta feature for prefix sharing (26)). To run SGLang and vLLM in a distributed fashion, we set up a load balancer that sends requests in a round-robin fashion to individual SGLang/vLLM instances (*i.e.*, non-prompt-aware data parallelism). As round- robin distributes requests evenly, these baselines capture a distributed serving system that balances request loads and then performs prefix sharing within each parallel instance.

Metrics. We use three key metrics: request per second, which measures serving capacity; average end-to-end request latency (including scheduling time, queueing time, prefill, and decoding time); and p99 request latency. Note that our metrics differ slightly from some existing LLM serving works (18; 48), as we do not use TPOT (time per output token) or TTFT (time to first token) as key metrics. This is because our

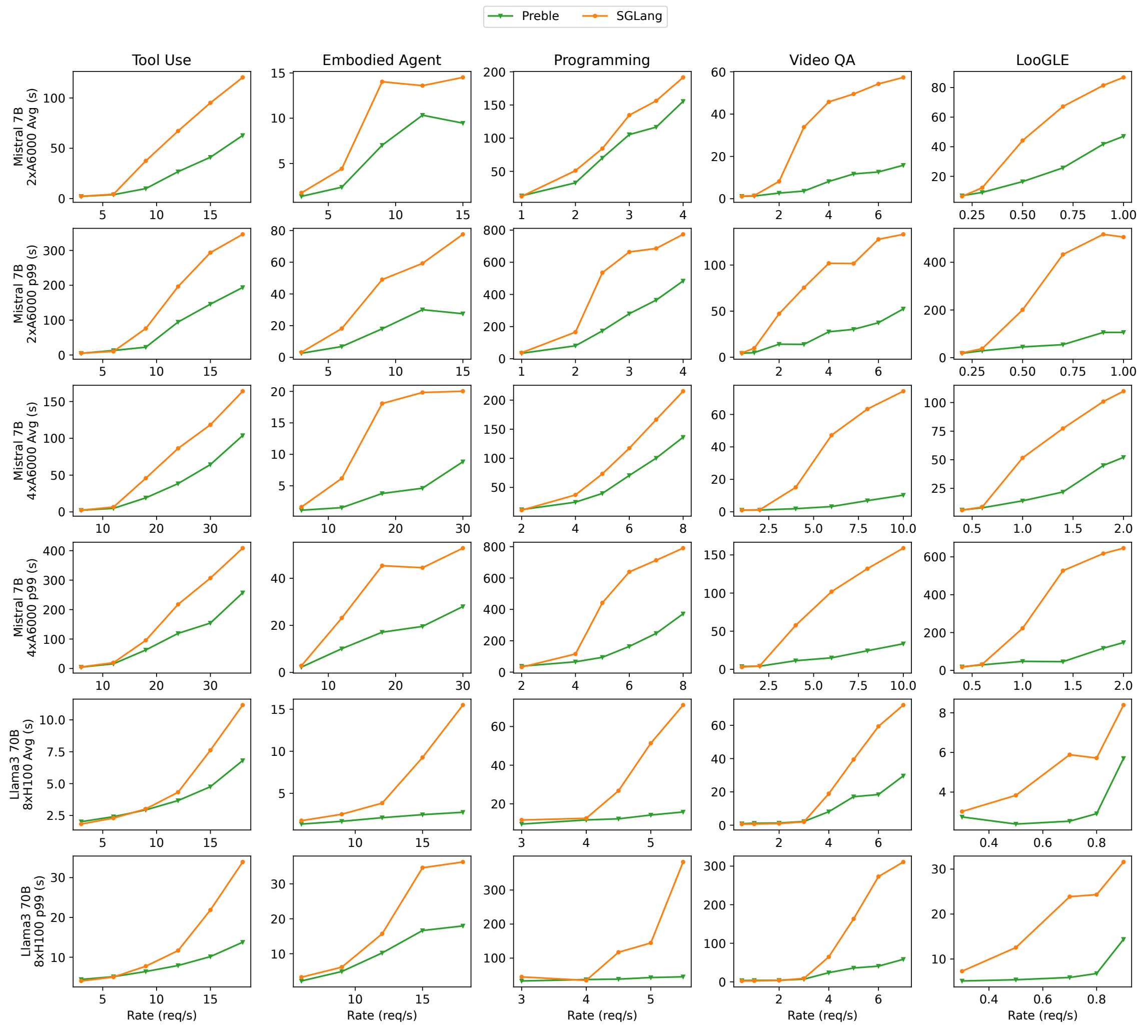

Figure 3: **End-to-end Workload Performance** *The top and middle two rows run on two and four A6000 GPUs with the Mistral 7B model. The bottom two rows run on eight H100 GPUs set up as 4-GPU tensor parallelism plus data parallelism with the Llama-3 70B model.*

target LLM use has short output lengths, rendering TPOT not as meaningful and TTFT close to the request latency. We consider p99 latency since it is important to control the tail latency in LLM serving as with all other user-facing services (7; 29).

4.3 End-to-End Workload Performance

We first present the overall performance of Preble and the baselines. Below, we focus on the comparison with SGLang as it is specifically designed for (single-GPU) prefix sharing while being up-to-date on major LLM serving techniques. We provide Preble's comparison to vLLM and to different SGLang versions in the Appendix C.

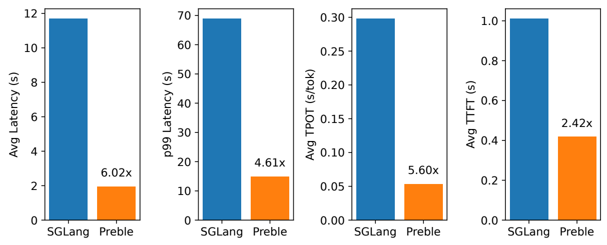

Figure 4: **Mixed Workload With Azure Trace** *Running Tool and Video mixed workloads with Azure trace arrival patterns on 4 A6000 GPUs.*

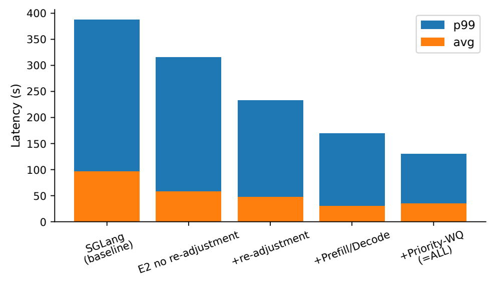

Figure 5: **Ablation Results** *Running ToolBench with Zipf-1.1 skew to different prompts running on four A6000 GPUs*

Single workload results. We now present the average and p99 latency against increasing requests arriving per second (RPS) of Preble and SGLang on the five workloads, two LLMs, and two GPU environments, as shown in Figure 3. Overall, Preble significantly outperforms the data-parallel SGLang baseline for all settings, as can be seen from Preble's lower average and p99 latency, especially under higher RPS (or the other way around, for the same latency target, Preble can serve higher RPS). Our improvements over SGLang range from 1.5× to 14.5× in terms of average latency and 2× to 10× in p99 latency.

Comparing across workloads, we see bigger improvements of Preble over SGLang on the Toolbench, embodied agent, video QA, and LooGLE workloads than the programming workloads. The programming workload has the longest decoding length among all the workloads. As decoding time starts to dominate total request latency, and we do not improve decoding performance, the room for improvement for Preble is smaller. Nonetheless, Preble still achieves 1.56× to 1.8× improvement in average latency and 3× to 4× in p99 latency over SGLang in the programming workload.

Comparing across the number of GPUs, Preble's relative improvement over the baselines stays similar when going from two to four A6000 GPUs. Considering absolute values, we see Preble successfully maintain similar latency even as RPS doubles, showing its strong scalability. When changing from A6000 to eight H100 and switching the Mistral 7B model to the Llama-3 70B model, we find relative improvements of Preble to increase.

Azure trace and mixed workloads. Our experiments above use a Poisson request arrival distribution (which is the same as most existing LLM works' experimental methodology (18; 22)). To understand Preble's performance under real-world request load, we run the tool use and video QA workloads using Azure's LLM request arrival pattern (Appendix A.6) instead of Poisson distributions. Here, we mix the two workloads to mimic Azure's mixed chat and code traces. As shown in Figure 4, Preble has significant improvements in average and p99 latencies and on average TTFT and TPOT.

4.4 Deep Dive

We now provide a detailed analysis of Preble, including an ablation study and global scheduler scalability test. Because of H100 GPUs' high cost and low availability, we run all experiments in this section with A6000 GPUs.

Ablation study. To understand where the benefits of Preble come from, we evaluate Preble by incrementally adding features presented in Section 3. We chose the tool use workload with a Zipf-1.1 popularity distribution among the prompts in the dataset to represent real-life skewed tool popularity. Other workloads and distributions benefit from a different set of techniques. We start with using the SGLang round-robin baseline. We first add the per-request E2 policy (Section 3.2), which results in an improvement on both average and p99 request latency because of E2's dynamic load partitioning. We then add the post-assignment global rebalancing and autoscaling, which successfully balances out load even more, resulting in further improvement, especially with p99. Further adding the prefill/decode-aware handling results in more improvement on both average and p99, since it considers the current batch composition and is able to better utilize the GPU resources. Finally, we add the local-scheduler priority-based wait-queue scheduling (§3.3), which, as expected, improves p99 but not average latency, as its goal is fairness.

Global scheduler performance and scalability. We measure the maximum throughput of Preble's global scheduler by sending a large number of requests (*e.g.*, 50,000) at once to eliminate the effect of request arrival patterns and saturate the scheduler. Since the global prefix tree search is the most time-consuming task at the global scheduler, we test the Toolbench and VideoQA workloads, which have the most complex and simplest prefix tree structures in our five workloads. Preble's global scheduler achieves a processing rate of 245 and 2931 requests per second for Toolbench and VideoQA. We also measure the network processing speed and find it not to be the bottleneck. With the peak GPU processing rate (30-150 tokens per second decoding speed with Mistral 7B on A100) and our workloads' output length (Table 1), one Preble global scheduler can sustain at least 70 to 391 concurrent A100 GPUs. If accounting for prefill time or running bigger models, our scheduler would sustain even more GPUs.

5 Related Works

LLMs' usages are shifting to be more prompt-heavy. As a result, the problem of prefill and decoding having different compute-to-memory ratios is exacerbated. Several recent works have targeted LLM usages with long prompts and proposed solutions to solve the imbalance problem (5; 4; 53; 32). The first approach, called *chunked prefill*, chunks a prompt and runs each chunk with other decoding requests in a batch in a single iteration to reduce or avoid waiting (5; 4). The second approach is to separate prefill and decoding to different GPUs to avoid prefill-decoding interference (32; 53). These solutions target long prompts but do not consider prompt sharing. Preble consider prompt length and sharing, and we use a novel sharing-based approach on top of chunked prefill to solve the prefill-decoding imbalance problem.

A recent work, SGLang (52), proposes to share prefixes across requests using a prefix tree. More recently, vLLM also added the support for prompt prefix sharing (26). Unlike Preble, SGLang and vLLM are both single-GPU solutions. To run them on a distributed GPU cluster, one would need to add a standard data or model parallelism layer and then run SGLang or vLLM on each GPU. As no parallelism or distributed serving systems today are prompt-aware, simply distributing requests or models and then performing prefix-sharing within a GPU ignores the cluster-level prefix-sharing opportunity. Apart from distributed support for long and shared prompts (Section 3.2), Preble also improves prefix-caching with fairness over SGLang and vLLM with a better waiting-request ordering policy (Section 3.3). Another recent work, Prompt Cache (9), proposes sharing arbitrary user-defined sub-sequences in a prompt by allowing mismatched positional encodings and incomplete attention computation. As such, non-prefix sharing is a lossy process that could result in lower-quality generation. Moreover, like SGLang, Prompt Cache is also a single-GPU solution and shares SGLang's limitations discussed above. Hydragen (17) is another recent work that proposes an efficient implementation of the attention operation for shared prefixes, which is orthogonal to Preble, as Preble can support any underlying attention kernels.

6 Conclusion

This paper identified the problem of distributed serving for long and sharing prompts. To solve this problem, we performed a study on five LLM workloads and one real LLM trace. We presented E2, a distributed LLM request scheduling algorithm targeting LLM usages with long and shared prompts. We built Preble, a distributed LLM serving system on top of the E2 algorithm and a hierarchical scheduling architecture. Our results show that Preble significantly improves LLM serving performance over SOTA serving systems while ensuring request fairness and controlling the tail latency.

References

[1] Flashinfer, October 2024.

[2] Prompt caching with claude. https://www.anthropic.com/news/prompt-caching, 2024.

[3] SGLang, October 2024.

[4] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, Santa Clara*, 2024.

[5] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, August 2023.

[6] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, Texas, November 2022. IEEE.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, oct 2007.

[8] Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Los Alamitos, CA, June 2020.

[9] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of the 7th MLSys Conference*, Santa Clara, CA, May 2024.

[10] Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. *arXiv preprint arXiv:2403.07714*, Mar 2024.

[11] Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. In *Advances in Neural Information Processing Systems 36*, New Orleans, Louisiana, December 2023.

[12] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, December 2021.

[13] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, June 2024.

[14] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning, Honolulu*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. PMLR, 17–23 Jul 2022.

[15] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, October 2023.

[16] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. *arXiv preprint arXiv:2310.06825*, October 2023.

[17] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, May 2024.

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, Koblenz, Germany, October 2023.

[19] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. Loogle: Can long-context language models understand long contexts? *arXiv preprint arXiv:2311.04939*, November 2023.

[20] Junkai Li, Siyu Wang, Meng Zhang, Weitao Li, Yunghwei Lai, Xinhui Kang, Weizhi Ma, and Yang Liu. Agent hospital: A simulacrum of hospital with evolvable medical agents. *arXiv preprint arXiv:2405.02957*, May 2024.

[21] Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu, and Deheng Ye. More agents is all you need. *arXiv preprint arXiv:2402.05120*, February 2024.

[22] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA, July 2023.

[23] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *Proceedings of the 12th International Conference on Learning Representations, Vienna*, 2024.

[24] Meta. Meta llama 3. https://llama.meta.com/llama3/, 2024.

[25] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2024.

[26] Sage Moore and Zhouhan Li. Automatic prefix caching. https://github.com/vllm-project/vllm/issues/2614, March 2024.

[27] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, April 2024.

[28] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*, July 2023.

[29] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[30] OpenAI. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/, September 2024.

[31] OpenAI. Video generation models as world simulators. https://openai.com/index/video-generation-models-as-world-simulators, 2024.

[32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[33] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*, Jul 2024.

[34] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations, Vienna*, 2024.

[35] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems, New Orleans*, 2023.

[36] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. {ALFW}orld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations, Virtual*, 2021.

[37] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: KV-cache streaming for fast, fault-tolerant generative LLM serving. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 46745–46771, Vancouver, Canada, 21–27 Jul 2024. PMLR.

[38] Neal Vaidya, Nick Comly, Joe DeLaere, Ankit Patel, and Fred Oh. Nvidia tensorrt-llm supercharges large language model inference on nvidia h100 gpus. https://developer.nvidia.com/blog/nvidia-tensorrt-llm-supercharges-large-language-model-inference-on-nvidia-h100-gpus/, 2023.

[39] Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. In *The Twelfth International Conference on Learning Representations*, Vienna, Austria, May 2024.

[40] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, New Orleans, LA, November 2024.

[42] Wikipedia. Radix tree. https://en.wikipedia.org/wiki/Radix_tree, 2024. [Online; accessed 1-October-2024].

[43] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, May 2023.

[44] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents, Vienna*, 2024.

[45] Junbin Xiao, Xindi Shang, Angela Yao, and Tat-Seng Chua. Next-qa: Next phase of question-answering to explaining temporal actions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9777–9786, June 2021.

[46] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Thirty-seventh Conference on Neural Information Processing Systems*, volume 36, New Orleans, Louisiana, December 2023.

[47] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, Kigali, Rwanda, April 2023.

[48] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, Carlsbad, CA, July 2022.

[49] Shoubin Yu, Jaemin Cho, Prateek Yadav, and Mohit Bansal. Self-chained image-language model for video localization and question answering. In *Thirty-seventh Conference on Neural Information Processing Systems*, New Orleans, Louisiana, December 2023.

[50] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. In *The Eleventh International Conference on Learning Representations*, Kigali, Rwanda, May 2023.

[51] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, New Orleans, Louisiana, December 2023.

[52] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, December 2023.

[53] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distllm: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*, Santa Clara, CA, July 2024.

7 Appendix

A Study on LLM Prompts

Today's LLM usage goes beyond simple chatting. As LLM usage becomes more commercialized, LLM prompts become more structured and complex, outshadowing the text an LLM generates. This section presents our study results of five popular new LLM use cases: tool (or API, agent) use (35), interacting with virtual environments as an embodied agent (11; 14), software program generation (28), answering questions about videos (45), and answer questions about long documents (19). Figure 8 demonstrates the prompt usages of these workloads. We study each case with real public datasets and understand their prompt features from a systems perspective. For datasets that do not provide outputs, we use Llama-3 7B model as the LLM to generate outputs. For each dataset, we construct a prefix tree for all the requests in the dataset (*i.e.*, assuming an infinite prefix cache).

Table 1 and Figure 6 summarize our study results, including prompt and decoding (output) length, amount of sharing in a prompt, key portion size in a prompt, and number of requests sharing a key portion. We define the "key portion" of a request as the deepest node in a path that has more tokens than the sum of its predecessors.

To understand real-world LLM user request features, we study a recently released public cloud LLM trace. This section ends with our summary insights.

Workload	Prompt Len	Output Len	Shared Prefix in Prompt	KeyPort. in Prompt	Req Share KeyPort.
Toolbench	(1835, 742)	(43, 16)	(85%, 13%)	(76%, 16%)	(39, 64)
Embodied Agent	(2285, 471)	(16, 13)	(97%, 14%)	(76%, 12%)	(48, 8)
Programming	(3871, 1656)	(190, 343)	(97%, 7.4%)	(78%, 13%)	(126, 2157)
Video QA	(9865, 5976)	(4, 1.5)	(88%, 32%)	(99%, 0.2%)	(8.6, 2)
LooGLE	(23474, 6105)	(16, 9.9)	(91%, 24%)	(94%, 15%)	(18, 8.6)

Table 1: **LLM Prompt Properties** *Each cell except for number of requests shows (mean, standard deviation). Length represented using number of tokens. "KeyPort." stands for Key Portion.*

A.1 Tool Use

Today, LLMs are often augmented by various tools such as calculators and web searches. To equip a model with the ability to invoke a tool, it must be given the correct syntax for querying the tool, along with examples (or "demonstrations") of tool use. We evaluate the Toolbench (10) dataset, which consists of more than 210k queries that call over 16k unique tools. Each query shares the same system prompt followed by tool-specific instructions. The final part of the query is the user's specific question or task. These are all concatenated together to form the final prompt. We find that most of the sharing comes from queries that all share the same tool, and these instructions can be 43x longer than the output length. The Toolbench workload is also representative of other tasks that "prep" an LLM in a similar fashion. For example, instead of tool-calling, LLMs can have roles layered on top of the system prompt, which is popular in emerging systems that utilize the same LLM with multiple roles to create an ensemble (44; 21; 20).

A.2 Embodied Agents

LLMs are increasingly found in agents that can interact with environments, such as a player in a role-playing game or controlling a robot. In this scenario, the LLM receives feedback from the environment, forms an action, and then "performs" the action. This is conducted in a loop until the model has achieved the goal. The workload we utilize is sourced from the ALFWorld (36) dataset and has 7.5k requests. Prompts first describe the environment and the task, followed by a demonstration of steps to solve the task. The model

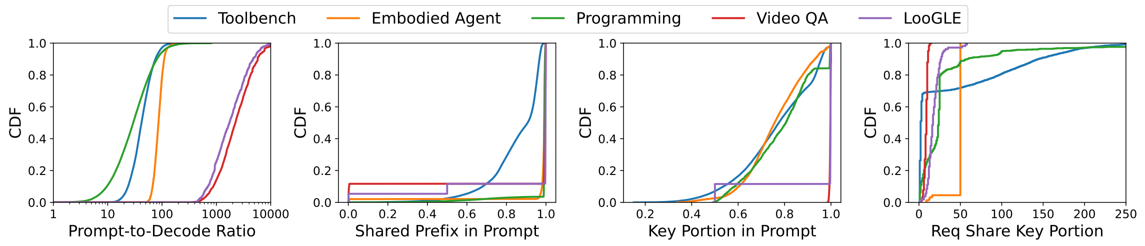

Figure 6: **CDF Plot of Key Metrics** *Showing CDF for all five workloads on prompt-to-decode ratio, shared prefix*

then solves its given task by looping over a planning step followed by an action step. After each action, the text-based environment returns an observation that the model incorporates into its next planning step. Every new invocation to the LLM in this loop is treated as a new request, resulting in each step sharing the context of previous steps. Interestingly, the number of steps is determined by LLM generation, creating an unpredictable sharing pattern. Because steps are chained together, prompts are still 157x longer than output tokens.

The embodied agent workload can represent a wide variety of other use cases, such as chain of thought (47; 41), multi-turn tool usage (39; 34), and chatbots (51). Any dependency between the model and the outside environment can be considered an agent receiving feedback.

A.3 Program Generation

One of the popular uses of LLMs is to generate software programs (28). We study the APPS competitive programming dataset (12), a dataset of programming problems. To generate better-quality programs, an approach taken by a recent paper (17) is to add a demonstration of several generic code examples before the user problem to instruct an LLM. This added demonstration is the same across all problems and becomes the system prompt. Following the system prompt is the programming problem description. Afterward, this approach invokes the LLM several times in parallel to generate multiple candidate programs, out of which the best is chosen to return to the user. As generated code is relatively long (compared to outputs of other workloads we study), the prompt-to-output ratio (20x) is relatively low. Prompt sharing comes from two places: the system prompt of code demonstration is shared across all requests, and the programming problem is shared across all parallel generations. Depending on how complex the problem is, its description could be longer or shorter than the system prompt; a problem description can also be partially the same as another problem description. Such complexity results in competitive programming having diverse key-portion properties. Such example demonstration and parallel generation technique is common in recent prompt engineering, for example, with ReAct (47), Tree-of-Thoughts (46), and Self Consistency (40).

A.4 Video Question and Answer

The advent of video models like OpenAI Sora (31) has created an explosion of interest in multi-modal models. The use of LLMs, then, goes beyond natural language. A recent usage is to answer questions about videos by tokenizing a video segment and inputting it to an LLM (49; 8). To study this, we analyze the NExT-QA benchmark (45), which consists of 8.5K questions for 1000 video segments. Prompts to the LLM consist of a tokenized video followed by a multiple-choice question. Because of the multiple-choice nature, the outputs of this dataset only have six tokens. Long tokens for representing videos plus short outputs result in this dataset having the highest prompt-to-decoding token ratio of all workloads we explored, with nearly 2500× more prompt tokens. Apart from videos, images and audio can also be tokenized to have LLMs answer questions, and we expect them to have similar properties as video QA.

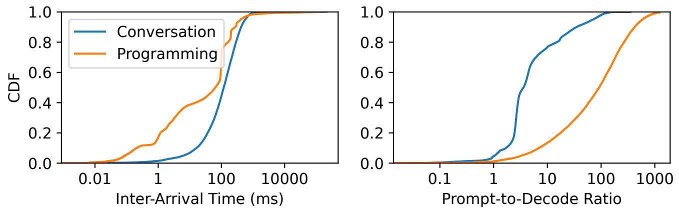

Figure 7: **Azure LLM Trace Analysis Results.**

A.5 Long Document Question and Answer

With newer models, the maximum context length has increased substantially (27; 23; 15), with the latest development supporting 1M tokens (27). Longer contexts enable new LLM applications such as asking questions about a long document or even a book. We evaluate this usage with the LooGLE dataset (19), a collection of 776 long documents and over 6.4k questions. LooGLE has a small system prompt of 13 tokens followed by a long document and then a question about the document. As a common practice, a user or multiple users often ask multiple questions to the same document, resulting in large amounts of shared tokens. Meanwhile, the answers are usually short (*e.g.*, a true or false). These features result in high prompt-to-decode ratio and high sharing ratio in LooGLE.

A.6 LLM Usages in the Wild

To understand LLM usage in the wild, we analyze the recently released Azure LLM Inference Trace (32). The trace includes two types of LLM usages: program generation and chat conversation. It provides request arrival time, prompt length, and decode length. As it does not provide actual request content, it is not feasible for us to evaluate prompt content or sharing. Figure 7 plot our analysis results in CDF. We find that the arrival rate is approximately 5 requests per second for chat conversation and 7 requests per second for programming. On average, chat requests arrive 118 ms apart while programming requests arrive 63 ms apart. The mean prompt-to-decode ratio for chat conversations is 4. Since we have no details about shared context from follow-up conversations, this number is expected to be much lower. For the longest 20% of all chat prompts, the mean prompt-to-decode ratio is 175, which is consistent with our observations on other workloads. For programming, the mean prompt-to-decode ratio is 92 for all prompts. This falls within the range of all the workloads we evaluated.

A.7 Summary Insights

Our analysis of the five real-world LLM workloads and a real user LLM request trace reveals several key findings.

Insight 1: Contrary to popular belief, prompts are significantly longer than output lengths because LLMs support longer context and new LLM usages keep emerging. We believe this trend will continue as LLMs are augmented with more capabilities. **Implication 1:** Optimizing prefill computation can largely improve overall application performance, and imbalanced prefill and decoding computation features should be considered in LLM serving.

Insight 2: Prompt sharing, or reuse, is common, and the sharing amount is high. Sharing can come from different user requests needing the same tools or instructions to solve a task. It can come from a user asking multiple questions about the same document or video. Context sharing can also happen within the same user task that is solved with a chain or a tree of steps. **Implication 2:** Reuse computation across shared prefixes can largely improve real workloads' performance and should be efficiently supported by distributed LLM serving systems.

Insight 3: Most requests have a portion of the prompt sequence that gets a different degree of sharing and is longer than its prefix, reflected as a key portion in prefix trees. Key portions account for the majority of prompts and are shared by a significant amount of requests. **Implication 3:** Identifying the key portion of prompts and optimizing the placement of requests according to their key portions is a viable way of reducing the complexity of scheduling while achieving good performance.

Insight 4: Real-world LLM usages have varying load intensity, and different usages (programming vs. conversation) have different loads. Real-world prompts are also much longer than decoding length, but different usages have different prompt-to-decode ratios. Still, the longest prompts are significantly longer. **Implication 4:** An efficient LLM serving system should consider complex, mixed-usage scenarios and factor in both load and prompt sharing variations.

B Prefill/Decoding Times

The prefill and decoding stages exhibit different computation behaviors, with the former being computation-bound and the latter being memory-bandwidth bounded. To understand their behaviors and to acquire prefill/decoding computation time functions to be used by E2, we profile the prefill and decoding stage performance with Mistral 7B on the A6000 GPU. Figure 9 plots the prefill time and its breaking downs when prompt length increases. As seen, longer prompts increase prefill time, suggesting that the more savings we can get from prefix sharing, the lower prefill time will be. Moreover, since the linear layer dominates the model forwarding at the prefill stage, the prefill time is overall linear to the prompt length. Figure 10 shows the performance of a single request's decoding performance with varying context lengths (the length of the prompt sequence plus the sequence generated thus far). We observe a similar linear relationship to context token length. Overall, these profiling results suggest that attention computation is regular. Thus, we could use the token length with a profile regression function to estimate computation time.

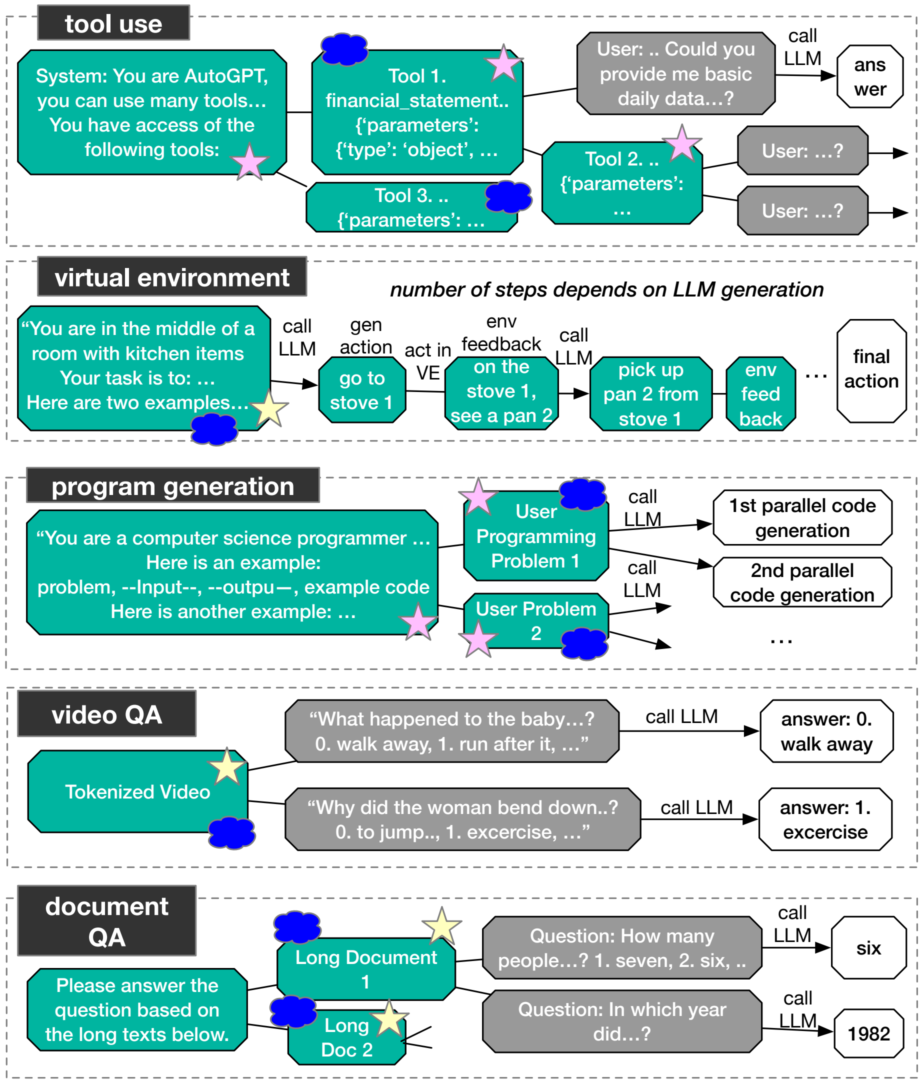
tool use
System: You are AutoGPT, you can use many tools… You have access to the following tools:
Tool 1. financial_statement.. {'parameters': {'type': 'object', …
Tool 3. .. {'parameters': …
User: .. Could you provide me basic daily data…?
call LLM
ans wer
Tool 2. .. {'parameters': …
User: …?
User: …?
virtual environment
number of steps depends on LLM generation
"You are in the middle of a room with kitchen items Your task is to: … Here are two examples…
call LLM
gen action
go to stove 1
act in VE
env feedback
on the stove 1, see a pan 2
call LLM
pick up pan 2 from stove 1
env feed back
…
final action
program generation
"You are a computer science programmer … Here is an example: problem, --Input--, --outpu—, example code Here is another example: …
User Programming Problem 1
User Problem 2
call LLM
1st parallel code generation
call LLM
2nd parallel code generation
call LLM
…
video QA
Tokenized Video
"What happened to the baby…? 0. walk away, 1. run after it, …"
call LLM
answer: 0. walk away
"Why did the woman bend down..? 0. to jump.., 1. excercise, …"
call LLM
answer: 1. excercise
document QA
Please answer the question based on the long texts below.
Long Document 1
Long Doc 2
Question: How many people…? 1. seven, 2. six, ..
call LLM
six
Question: In which year did…?
call LLM
1982

Figure 8: **Workload Demonstration.** *Green boxes represent shared prefixes. Grey boxes are non-shared prompts. White boxes are output generation. Yellow star represents key portions that always happen at fixed parts; pink stars at non-fixed parts. Blue clouds represent the parts that would be used for distributing prefixes if knowing the oracle.*

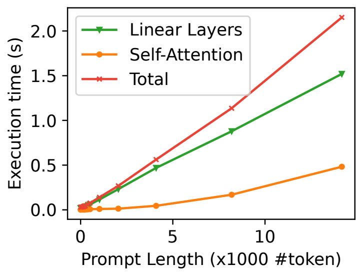

Figure 9: **Prefill Time Decomposition**

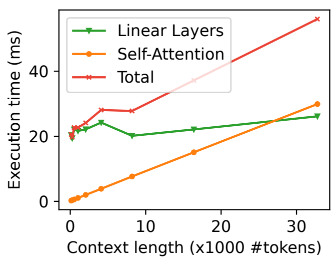

Figure 10: **Decoding Time**

Algorithm 1 E2 Global Scheduling Algorithm

```
function ScheduleRequest($R_k$)
    Match $R_k$ to global radix tree
    $cached\_len \leftarrow$ sum of matched length
    $missed\_len \leftarrow prompt\_len - cached\_len$

    if $missed\_len < cached\_len$ then                              ▷ Exploit $R_k$
        $K \leftarrow$ GPUs with longest node in matched path
        for each GPU $i$ in $K$ do
            $Cost_i \leftarrow$ LoadCost($i, R_k$)
        end for
        return $i$ with lowest $Cost_i$
    else                                                             ▷ Explore $R_k$
        for each GPU $i$ in all GPUs do
            $Ratio_i \leftarrow$ DecodeRatio($i$)
        end for
        ▷ ImbalR: calc based on GPU type and LLM
        if highest $Ratio_{max} >$ ImbalR then
            return $max$
        end if

        for each GPU $i$ in all GPUs do
            $Cost_i \leftarrow$ LoadCost($i, R_k$)
        end for
        return $i$ with lowest $Cost_i$
    end if
end function
```

Algorithm 2 GPU Load Cost Calculation

```
▷ Load cost calculation for GPU $i$ and request $R_k$
function LOADCOST($i$, $R_k$)
    $L \leftarrow 0$; $M \leftarrow 0$; $P \leftarrow 0$;

    ▷ Calculate total load on GPU $i$
    for each $R_j$ in history $H$ do
        $missed\_len \leftarrow$ non-cached prompt length for $j$
        $L \leftarrow L +$ PREFILLTIME$(missed\_len)$
        $decode\_len \leftarrow$ average request output length in $H$
        $L \leftarrow L +$ DECODETIME$(decode\_len)$
    end for

    ▷ Calculate eviction cost
    $E \leftarrow$ tree nodes to evict on GPU $i$ to run $R_k$
    for each $j$ in $E$ do
        $N_j \leftarrow$ number of requests sharing $j$ in $H$ / number of total requests in $H$
        $M \leftarrow M+$ PREFILLTIME(length of $j$) $\times N_j$
    end for

    ▷ Calculate cost to run $R_k$
    $missed\_len\_k \leftarrow$ non-cached prompt length for $R_k$
    $P \leftarrow$ PREFILLTIME$(missed\_len\_k)$

    return $L + M + P$
end function
```

Algorithm 3 E2 Local Scheduling Algorithm

```
function SCHEDULEREQUESTS
    for all requests in waiting queue do
        Determine prefix hit ratio
        Assign to priority group based on hit ratio
    end for
    Process groups in round-robin fashion with limits
end function
```

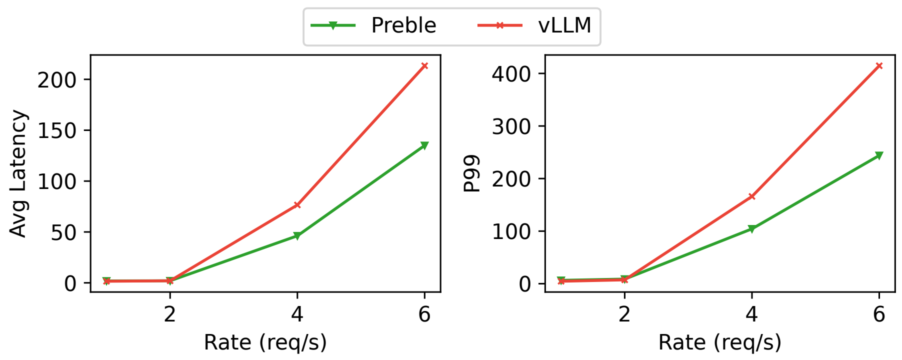

Figure 11: **vLLM Backend Performance** *Evaluated on the Video QA workload using the Mistral 7B model on 2 GPUs.*

C Comparison with vLLM and Other SGLang Versions

To demonstrate Preble's versatility with multiple LLM backends, we evaluate Preble on vLLM with the vanilla vLLM as the baseline. vLLM recently added support for prefix caching, which we include in the baseline. We use a slightly different version of the Mistral 7B model (v0.2) for this experiment, as vLLM only supports this version. Figure 11 plots the results of running the VideoQA workload on 2 GPUs and the Mistral 7B v0.2 model for both Preble and vLLM. Compared to SGLang as a backend, vLLM as a backend gives Preble less relative improvement for several reasons: 1) local-GPU prefix sharing is in beta version and not as performant as SGLang; 2) vLLM does not use the flash_infer kernel which makes prefix sharing more efficient; and 3) vLLM does not support chunked prefill together with prefix caching. 4) vLLM has significant scheduling delay

To demonstrate Preble's versatility across multiple SGLang versions, we evaluate the latest SGLang version (v0.3) (3) and the latest FlashInfer kernel (v1.6) (1) in addition to the v0.1.12 we used in Section 4. SGLang made two major changes between these two versions: 1) the new FlashInfer kernel improved the performance of sharing 32K context length in the same batch, and 2) SGLang reduced its scheduling overhead and applied other engine optimizations. Figure 12 plots the results of running LooGLE and Toolbench with SGLang v0.3 and Preble. Overall, Preble's improvements over SGLang persist across SGLang versions.

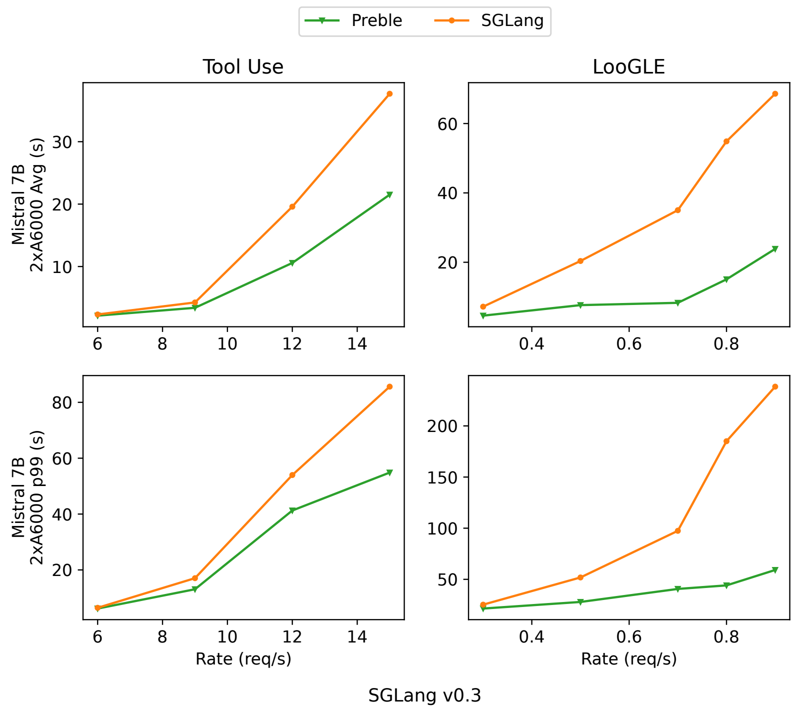

Figure 12: **Latest SGLang v0.3 and Flashinfer 0.1.6**

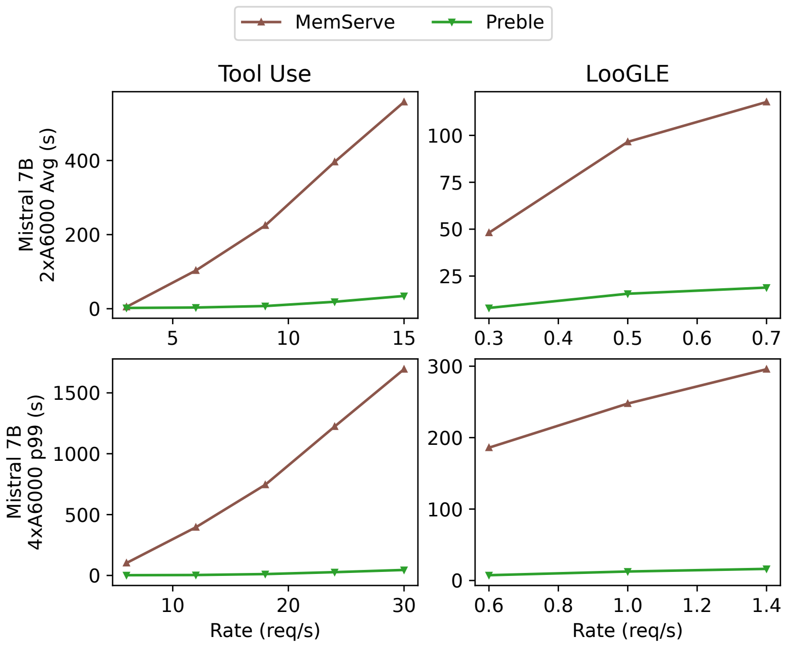

Figure 13: **MemServe Results** *Preble and MemServe performance running Tool and Video workloads on 2, 4 GPUs on A6000.*

D Comparison with Memeserve

We also provide a comparison against MemServe's scheduler in Figure 13. Preble significantly outperforms the scheduler due to MemServe's over exploitation of the prefix.

E Additional Scalability Experiments

In order to test Preble's scalability at a larger number of GPUs, we tested LooGLE using 8 GPUs and 16 GPUs. We provide the average latency in Figure 14. We also provide a plot showing the max sustainable RPS under 20 seconds of Preble and Round Robin baseline in Figure 15. Based on the results, we can see that there's a linear scalability as instances increase.

F Dynamic E2 Scheduling

We also provide an experiment with dynamically selecting exploitation vs exploration based on the total load of the cluster as well as token count in Figure 16. The results between the two approaches are relatively similar. In LooGLE dataset, the default Preble performs (min: -4%, max: 9.2%, avg: 3%) better than the knob based Preble. In the Toolbench dataset, the default Preble performs (min: -1.6%, max: 15%, avg: 5.43%) than the dynamic knob based Preble.

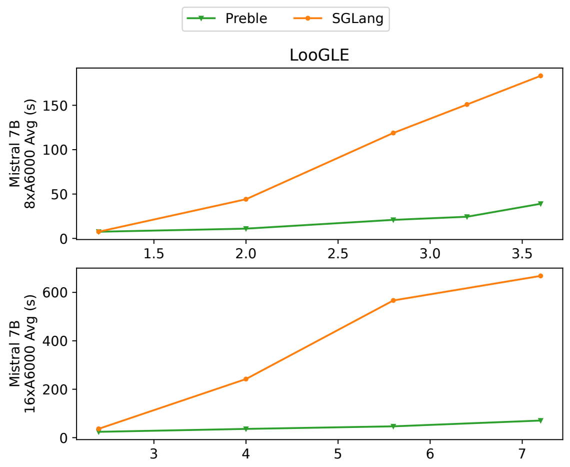

Figure 14: **Scalability Test with Larger GPU Clusters**

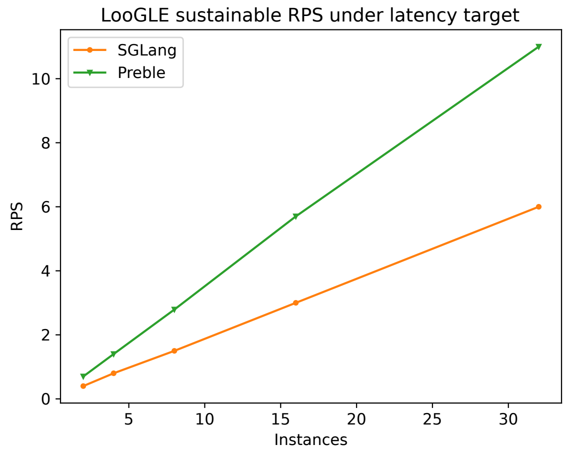

Figure 15: **Max sustainable RPS with a larger GPU cluster**

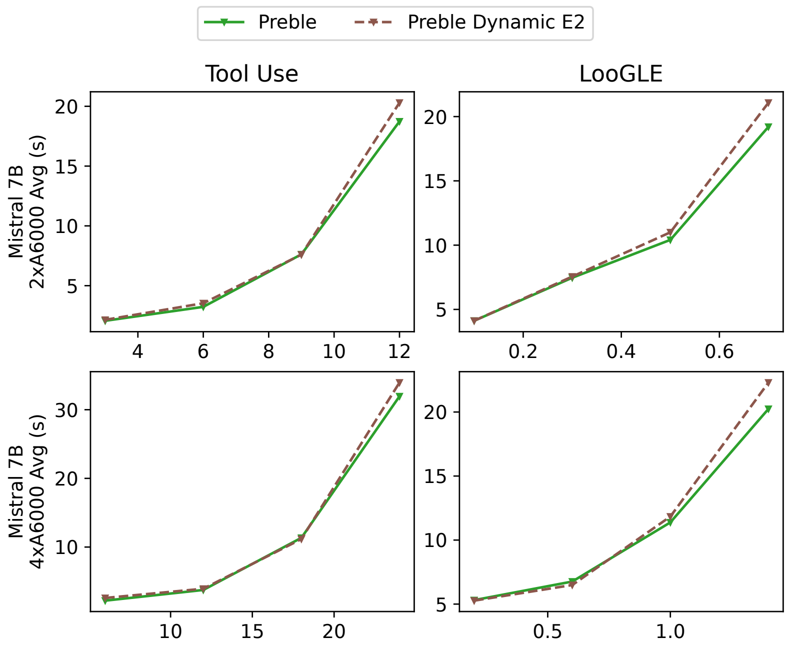

Figure 16: **Testing Dynamic E2 Algorithm with GPU Utilization as Knob**