PrefillOnly: An Inference Engine for Prefill-only Workloads in Large Language Model Applications

Kuntai Du[‖], Bowen Wang[†], Chen Zhang[†], Yiming Cheng[‖], Qing Lan[‡], Hejian Sang[‡], Yihua Cheng[‖], Jiayi Yao[‖], Xiaoxuan Liu[¶], Yifan Qiao[¶], Ion Stoica[¶], Junchen Jiang[‖]

[‖]University of Chicago [†]Tsinghua University [‡]LinkedIn [¶]UC Berkeley

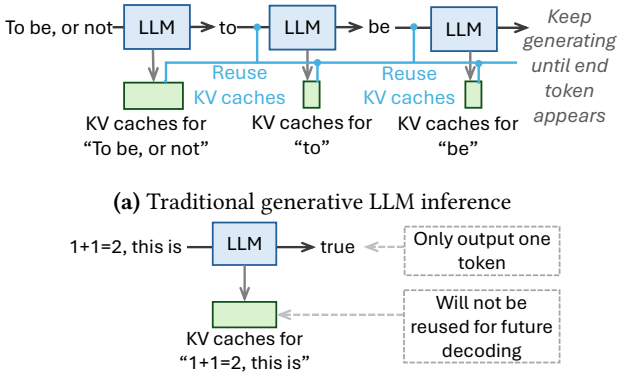
Abstract

Besides typical generative applications, like ChatGPT, GitHub Copilot, and Cursor, we observe an emerging trend that LLMs are increasingly used in traditional discriminative tasks, such as recommendation, credit verification, and data labeling. The key characteristic of these emerging use cases is that the LLM generates only a *single* output token, rather than an arbitrarily long sequence of tokens. We call this *prefill-only* workload. However, since existing LLM engines assume arbitrary output lengths, they fail to leverage the unique properties of prefill-only workloads. In this paper, we present PrefillOnly, the first LLM inference engine that improves the inference throughput and latency by fully embracing the properties of prefill-only workloads. First, since it generates only one token, PrefillOnly only needs to store the KV cache of only the last computed layer, rather than of all layers. This drastically reduces the GPU memory footprint of LLM inference and allows handling long inputs without using solutions that reduces throughput, such as cross-GPU KV cache parallelization. Second, because the output length is fixed, rather than arbitrary, PrefillOnly can precisely determine the job completion time (JCT) of each prefill-only request before it starts. This enables efficient JCT-aware scheduling policies such as shortest remaining job first. PrefillOnly can process upto 4× larger queries per second without inflating average and P99 latency.

1 Introduction

Nowadays, large language models (LLMs) are widely used in generative tasks, such as chatting (*e.g.,* ChatGPT [34], Character.AI [1]), code generation (*e.g.,* GitHub Copilot [14], Cursor [5]), and summarization (*e.g.,* Perplexity [37]), which generate new content (of variable lengths) for people to read and use. However, researchers and industry practitioners also recently started to use LLMs to replace traditional deep learning models in **discriminative** tasks, such as recommendation [12, 42, 44], credit verification [11, 41, 45], and data labeling [16, 25, 53]. The reason is two-fold:

- *Streamlining development*: Traditional deep learning solutions often require multiple iterations of data preprocessing [51], feature engineering [7], and model tuning [21], which involves cross-team collaboration and can be time-consuming. In contrast, LLMs can directly process raw data and are general enough to obviate fine-tuning [12, 13, 40], allowing developers to interactively debug and improve quality by just changing the prompts.
- *High decision quality*: By carefully choosing the LLMs and engineering the prompt of the LLM-based pipeline, LLMs can achieve comparable or even higher decision quality compared to production models [12].



(a) Traditional generative LLM inference

(b) Prefill-only LLM inference

Figure 1. *Contrasting traditional LLM inference and prefill-only LLM inference. A prefill-only request does not reuse its KV cache for long decoding as it only generates one output token.*

Prefill-only workloads: Our observation is that when used in these discriminative tasks, the LLM generates only *one single token as output* for each incoming request[24, 25, 30, 39, 43], because a single-token output is sufficient to express the output of these tasks, *e.g.,* a preferred choice or a label. For example, the following prompt is used in a document recommendation application: "`Here is the user profile: [user profile], and here is the document: [document]. Should we recommend this document to this user? Your answer is: `". To determine whether this document should be recommended, the LLM only needs to generate one output token (*i.e.,* `Yes` or `No`).[1]

[1]To ensure that LLM only generates `Yes` or `No`, the developers do not have to fine-tune the LLM or add extra prompts; instead, they can pass a list of acceptable tokens (*e.g.,* `Yes` and `No`) to the LLM engine so that LLM engine only samples output from this list. More discussion in §2.4.

In this paper, we refer to such workloads as *prefill-only*, because to generate one token, the LLM engine only needs to run the prefilling stage of LLM inference.

Untapped opportunities: Unlike traditional LLM requests, prefill-only requests present two unique opportunities.

- *Smaller active GPU memory footprint:* Normally, the LLM engine stores the KV cache of all layers in GPU memory to save repeated computation when decoding each output token. This amounts to a large amount of KV cache data, particularly when the input is long (*e.g.,* the user profile in the example above may contain months of user browsing history). For prefill-only requests, however, most stored KV cache values will not be reused, because the engine does not need to decode more than one token (as shown in Figure 1), making it possible to significantly reduce the active GPU memory footprint during the inference.
- *Less uncertainty in JCT:* A typical LLM request's job completion time (JCT) is hard to pre-determine because the output length could be arbitrary depending on the sampling[27, 38]. For prefill-only requests, however, the output length is always one, so the LLM engine has enough knowledge to determine the amount of work for each request, making better scheduling logic possible.

Unfortunately, existing LLM engines fall short of leveraging these properties of prefill-only workloads. Because these LLM engines assume the output length of a request can be arbitrarily long, they must store in GPU memory all the KV cache of all tokens at all layers, and their request scheduling logic must handle uncertainties on JCT, rather than using the classic efficient logics like shortest-job first.

***PrefillOnly* fully embraces the opportunities:** This paper presents PrefillOnly, the first LLM inference engine tailored for prefill-only workload. Its technical contribution is twofold.

First, it uses a novel *hybrid prefilling* to allow the LLM engine to keep only the KV cache of one layer, thus drastically reducing the active memory footprint during inference. This obviates the cross-GPU communication overheads and slowdown as each request can now fit in fewer GPUs. Yet, this is easier said than done. Naively keeping the KV cache of one layer does not directly reduce the memory footprint since the temporary tensors allocated during LLM inference still use a large amount of GPU memory, and while using chunked prefilling reduces the size of these temporary tensors, it forces the KV cache of all layers to remain in GPU memory between chunks.

Our hybrid prefilling addresses this conundrum by processing non-attention layers chunk-by-chunk but processing attention layers normally. The intuition is that most GPU memory usage in LLM inference comes from non-attention layers, and chunking them can significantly reduce the GPU memory footprint, as these layers are all linear layers, so each chunk can be computed independently with each other. We note that, though hybrid prefilling does not only benefit prefill-only requests, it is the enabler for PrefillOnly to potentially discard or offload the KV caches outside the GPU memory, as it ensures the prefilling is finished within one LLM forward pass rather than multiple passes that require reusing the KV caches between passes. Also, we emphasize that hybrid prefilling merely makes it possible to not keep all the KV cache in active GPU memory during the inference — the LLM inference engine can choose between keeping the KV cache in GPU memory, offloading part of it to CPU memory or discarding part of it.

Second, PrefillOnly's request scheduling logic leverages the fact that the system has enough knowledge to pre-determine each prefill-only request's completion time. This makes a range of classic JCT-based scheduling algorithms possible. However, this again is easier said than done. For each incoming request, its JCT also depends on whether its prefix's KV cache is available. Given that the stored KV caches can constantly change, PrefillOnly *continuously recalibrates* the JCT of waiting requests every time before running the scheduling algorithm. With the continuous JCT calibration, PrefillOnly can more precisely determine the JCT of each request for better scheduling *and* maximize the chance for KV caches of a prefix to be reused across requests (as cache-hit requests have lower JCT and will be prioritized, preventing irrelevant requests from evicting the prefix cache needed by cache-hit requests). Moreover, for better fairness and starvation avoidance, PrefillOnly also offsets the JCT based on request waiting time (as elaborated in §6.3).

While conceptually, the prefill-only properties are not unique to LLMs but also apply to many classic deep neural networks, the exact techniques are specialized to LLMs: hybrid prefilling leverages the property that the majority of LLMs only contain attention layers and linear layers, and PrefillOnly continuously calibrates the JCT based on LLM prefix KV caching.

In short, our contributions are:

- We identify the emerging trend that people use LLMs to substitute traditional deep-learning-based pipelines, and characterize a new workload – *the prefill-only workload* – underlying such trend.
- We present PrefillOnly, the first LLM inference engine that fully embraces the unique properties of prefill-only workloads, to drastically reduce active GPU memory footprints and enable better JCT-based scheduling.

We implement PrefillOnly on top of an enterprise-grade, state-of-the-art LLM inference engine (vLLM [22]). For the generalizability of PrefillOnly, we implement hybrid prefilling via `torch.compile` so that PrefillOnly can generalize to different LLM models, and implement logics related to KV cache storage without modifying hardware-related kernels

so that PrefillOnly generalizes to different hardware platforms. We evaluate PrefillOnly under 4 hardware setups, 3 LLM models, and 2 traces. Our evaluation shows that PrefillOnly can handle upto 4× query-per-second while still achieving lower average and P99 latency than the baselines.

2 Motivation

2.1 Preliminary of large language models

This section introduces the basic concepts in large language model (LLM) inference.

Prefilling and decoding: LLM inference contains two phases. The prefilling phase of LLM processes the input and generates one output token. Then, the LLM engine runs multiple rounds of the decoding phase, where each decoding phase forwards the request through the LLM and generates one more output token.

KV caches: As LLM inference incurs multiple rounds, when processing one request, the LLM engine will store the intermediate tensors produced by the attention layers inside the GPU, known as KV caches, to accelerate future rounds of LLM inference. The size of KV caches can be large—*e.g.,* the KV cache size of a request with 100,000 tokens is around 12 GB for a medium-sized LLM model (Llama-3.1-8B).

Prefix caching: The KV caches generated by one request can be reused by another request if they share part of the prefix, which is commonly referred to as prefix caches. As a result, existing LLM inference engines [23, 56] will not immediately free these KV caches after request execution, but instead cache them in the GPU so that future requests can potentially reuse the KV caches.

2.2 Substituting traditional deep learning models with LLMs

In addition to generative LLM applications that generate new content for people to read and use, (*e.g.,* ChatGPT [34], Character.AI [1], GitHub Copilot [14], Cursor [14], Perplexity [37] and more), both industry and academia have started to use LLMs to replace traditional deep learning pipelines in applications such as recommendation, credit verification, and data labeling mainly for two reasons.

LLM streamlines the development: Developing traditional deep learning pipelines is time-consuming and complex, as it requires co-optimizing multiple stages in the pipeline, from data cleaning to model fine-tuning, which requires cross-team collaboration and extensive infrastructure support, and eventually accumulates maintenance debt. In contrast, LLM can take raw text as input and chat with the developer, and is general enough to obviate fine-tuning. This allows the developer to interactively debug and improve the pipeline by just chatting with the LLM. This argument is supported by recent literature [12], which shows that a single LLM model can serve over 30 tasks across over 8 different domains.

LLM achieves higher decision quality: Further, by properly selecting the size of the LLM model and engineering the prompt, LLM can generalize to out-of-domain tasks and surfaces, and achieves comparable performance similar to or better than a production model [12].

2.3 Underlying workload: prefill-only workload

We observe that people use LLMs in these applications in a different way from those generative applications. Concretely, these applications only let the LLM generate one single output token for each request. We call these requests *prefill-only requests* (as they only require the LLM engine to run prefilling to generate a single output token), and name such a workload a prefill-only workload.

Single token suffices to express LLM's preference: Generating one single token still allows LLM to express its preference between different options. To illustrate this, we use post recommendations as an example. Assume that we are a social media platform that aims to recommend social media posts to a specific user via LLM. In this case, the recommendation system will first gather user's browsing history as the profile. Then, the system selects, for example, fifty posts that the user might be interested in using traditional heuristics like embedding-based similarity search. Then, the recommendation system sends fifty LLM requests to the LLM, one per document. Here is an example of such LLM request:

> You are a recommendation assistant to use user's profile and history to recommend the item that the user is most interested in. Here is the user profile:
> [User profile]
> Here is the browsing history of an user:
> [User browsing history]
> If we recommend the following article to this user, will the user be interested in reading it? Please response using Yes or No.
> [Article]
> Your answer is:

Then, we constraint the output of LLM to only `Yes` and `No`, and let LLM prefill this request to yield two probability numbers: $\mathbb{P}$(`Yes`) and $\mathbb{P}$(`No`), where their sum equals to 1. After that, the recommendation system will use $\mathbb{P}$(`Yes`) as the recommendation score.

Lower latency: Generating one single token also significantly reduces LLM inference latency. This is because the input processing of LLMs is much faster than output generation. Concretely, using Llama 3.1-8B-model and one NVIDIA H100, we measure that handling a request with 2048 tokens input and 256 tokens output is 1.5× slower than handling a request with 2048 tokens input and 1 token output.

Clearly-defined and controlled output behavior: Further, we argue that prefill-only workload allows the developer to simply define and control the output behavior by

passing over a list of acceptable tokens to the LLM engine and then let the LLM engine to sample only output from this list. In contrast, it is difficult to clearly define and guarantee the expected LLM output behavior in traditional generative requests, which motivates a long line of research (*e.g.,* [8, 49]).

2.4 Characteristics of prefill-only workload

We conclude two characteristics of prefill-only workload.

First, the input length of prefill-only requests is typically long. For example, in a documentation recommendation application, the user profile potentially contains months of the user's browsing history, which can easily reach tens of thousands of tokens, and even more.

Second, different from a traditional LLM workload that is GPU-memory-bound, a prefill-only workload is bounded by GPU computation.

Sharing GPU resources with traditional generative workload is impractical: To meet the stringent application requirements, instead of sharing the GPU with traditional generative requests, one needs to allocate GPU dedicated to prefill-only workloads. This is because the inter-workload interference of LLM applications is significant. For example, serving both prefill-only requests and traditional generative requests on the same GPUs may greatly increase the average and P99 time-per-output-token in generative use cases, as the decoding jobs will be batched with prefill jobs more frequently compared to not mixing these two workloads.

Further, the volume of prefill-only workload is large enough that it deserves dedicated GPU resources. For example, in recommendation workload, the typical queries per second is at the scale of tens of thousands, which demands hundreds or even thousands of H100 GPUs to serve.

2.5 Limitation of existing LLM engines

In practice, we observe two issues that limit the capacity of existing LLM inference engines in prefill-only workloads.

Trade throughput to handle long requests: As existing LLM engines fully store the KV caches, this limits the maximum input length (MIL in short) that they can handle, as the size of KV caches linearly scales with respect to input length. For example, the MIL is only 11,000 tokens on NVIDIA A100 40GB GPU with Qwen-32B model (FP8-quantized).

As a result, when the expected maximum request length exceeds MIL, the LLM engines have to choose one of the following approaches, and all of them come at the cost of reduced throughput.

- *Chunked prefilling*. Chunked prefilling prefills the request *chunk-by-chunk* so that the LLM engine can handle longer requests without parallelizing the inference. However, this technique reduces attention kernel performance, resulting in lower throughput. Concretely, we measure that chunked prefill will lower the end-to-end throughput by 14% when chunking the input length of 20,000 with a chunk size of 512. Further, this technique can only marginally increase the context length by less than 2× (as shown in Table 2), as it forces the LLM engine to fully store the KV caches of all previous chunks.
- *Tensor parallelism.* One can increase the MIL by tensor parallelism. However, although tensor parallelism can reduce the latency when used together with high-speed interconnection hardware such as NVLink, it may instead inflate the latency when high-speed interconnection is not available, and it always decreases the overall throughput. This is because tensor parallelisms require expensive all-reduce communication between GPUs, which takes a significant portion of GPU time, even with the acceleration of NVLink and resulting in GPU computation being under-utilized during the all-reduce communication.
- *Pipeline parallelism.* One can also increase the MIL by pipeline parallelism. Ideally, pipeline parallelism has the same latency-throughput trade-off as without parallelization on the same number of GPUs. However, pipeline parallelism introduces bubbles when the request length varies, and thus has sub-optimal latency-throughput trade-offs. Such bubbles can be minimized by chunking the input to the same chunk size via chunked prefilling, but chunked prefilling itself hurts the latency-throughput trade-offs as mentioned before.

We will further discuss parallelization-based solutions in prefix caching scenario (§5.2).

Scheduling algorithm is unaware of JCT: Existing LLM engines typically leverage JCT-agnostic scheduling, such as first-come-first-serve scheduling, as the output length of LLM requests can be non-deterministic and difficult to predict. If the JCT can be accurately estimated, one can leverage JCT-aware scheduling (like shortest remaining job first) to further reduce the latency of requests.

2.6 Opportunity and challenges

In prefill-only workload, we characterize two optimization opportunities:

- The active GPU memory of prefill-only inference is much lower — the generated KV caches will not be reused for future decoding, so it can potentially be offloaded or discarded from the GPU (as shown in Figure 1). This allows one to potentially handle much longer requests without using solutions that degrade the throughput, like parallelizing the KV caches or chunking the input.
- The JCT of prefill-only requests is deterministic and predictable as the output length is always 1.

A naive solution to leverage the opportunities above is by offline profiling how the JCT changes with respect to request length, and in the online phase, the LLM engine uses this JCT profile to obtain the JCT of each incoming request, and then schedules the waiting requests using JCT-aware algorithms

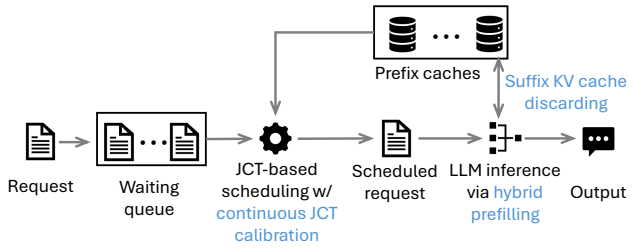


Figure 2. *Overview of PrefillOnly and its core techniques.*

(e.g., shortest remaining job first), and then discards the KV caches during inference. This solution is widely adopted in traditional deep learning systems, but it has two severe limitations:

Simply dropping KV caches increases MIL marginally: Ideally, dropping the KV caches could allow the LLM engine to handle up to the number-of-layers-time longer input length. However, in practice, we observe that dropping the KV caches only yields 1.6× MIL improvement, measured using NVIDIA L4 using Llama-3.1-8B model. This is because the LLM inference itself allocates large temporary tensors that consume a significant amount of GPU memory and limit MIL. See §4.1 for more analysis.

Incompatible with prefix caching: Existing LLM engines store the KV caches in the LLM engine so that future requests with the same prefix can be accelerated. This technique is known as prefix caching. Fully discarding the KV caches prevents such optimization.

3 Overview

In order to improve the latency-throughput trade-off in prefill-only workload, we propose PrefillOnly, the first inference engine tailored for prefill-only workload, built on top of production-level serving engine, vLLM.

3.1 Overview of PrefillOnly

PrefillOnly is an inference engine that serves prefill-only requests online. The workflow of PrefillOnly is as follows.

Profile run: PrefillOnly assumes that the user provides the maximum request length (MIL) PrefillOnly needs to handle. Based on user-provided MIL, PrefillOnly calculates the amount GPU memory required for LLM inference, by forwarding a fake request with maximum request length through the LLM model and measure the peak GPU memory usage during this period. The remaining GPU memory is then used as the space for prefix KV caches.

During runtime: PrefillOnly opens an HTTP server compatible with the OpenAI API protocol for the user to send their prefill-only requests. When a new request arrives, PrefillOnly tokenizes the request and sends it to the waiting queue of the scheduler process using ZeroMQ-based RPC. The scheduler process then schedules the requests in the granularity of *step*. During each step, PrefillOnly enumerates the requests in the waiting queue to find the request with minimum JCT (more details in §3.2). Then, PrefillOnly pops out this request and sends it to the executor processes. Finally, the request executor processes then execute the request (more details in §3.2) and return the prefill-only probability score all the way back to the user.

We summarize the workflow of PrefillOnly in Figure 2.

3.2 Core techniques of PrefillOnly

At the core of PrefillOnly is the following techniques that maximally increase MIL and throughput of prefill-only requests (we illustrate how these techniques participate in the workflow in Figure 2):

- *Hybrid prefilling*. PrefillOnly prefills non-attention layers chunk-by-chunk, but prefills the attention layers normally. This technique allows PrefillOnly to maximally increase the MIL by simultaneously chunking the intermediate tensors of non-attention layers and discarding KV caches for attention layers, while not hurting the performance of attention operations.
- *Suffix KV cache discarding / offloading*. PrefillOnly discards / offloads the KV caches of suffix tokens when the KV caches cannot fit into the GPU, allowing PrefillOnly to handle much longer requests without using solutions (chunked prefill or parallelizing the inference) that create extra overhead and degrade throughput. In this paper, for simplicity, PrefillOnly discards the useless KV cache, though in practice PrefillOnly can also choose to offload the KV cache to CPU memory if such an offloading mechanism is available.
- *Continuous JCT calibration*. PrefillOnly continuously re-estimates the JCT of each request based on what requests are previously scheduled, and then schedules just one request with the lowest JCT. This allows PrefillOnly to obtain accurate JCT estimation using what requests are previously scheduled, which allows PrefillOnly to increase the prefix cache hit rate by dynamically identifying the requests that can hit the cache created by the newly-scheduled requests, and thus reduces the request latency.

4 Hybrid prefilling

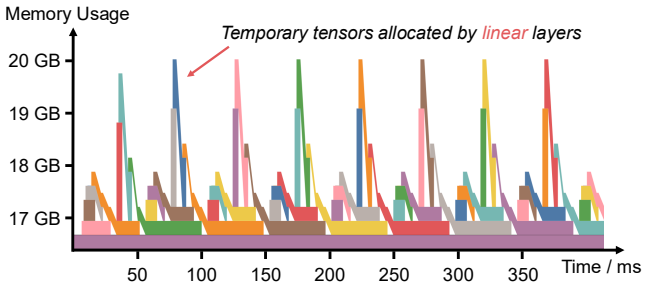
In this section, we first introduce hybrid prefilling, the key optimization for PrefillOnly to improve MIL by maximally controlling the GPU memory footprint during LLM inference.

4.1 Bottleneck: intermediate tensors of linear layers

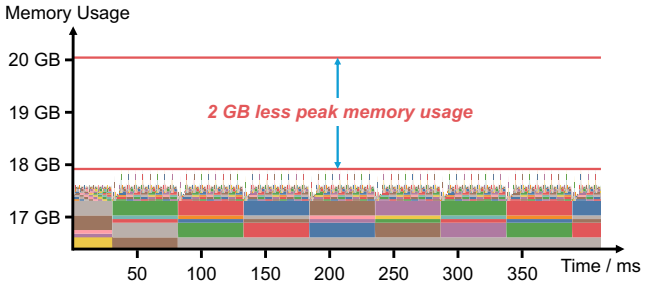
As mentioned in §2.5, simple solutions only marginally increase MIL: storing the KV cache of active layer increases MIL by merely 1.6×, while chunked prefilling increases MIL

by 2× but at the cost of reducing the kernel performance of attention operation, thus reducing throughput.

To understand why simply discarding the KV cache of active layers does not work well, we profile how the GPU memory usage of PyTorch GPU memory allocator varies over time when prefilling a request with 32, 768 tokens, using the Llama-3.1-8B model.



(a) Without hybrid prefilling



(b) With hybrid prefilling

Figure 3. *GPU memory traces of prefilling 32,768 tokens through Llama-3.1-8B model. intermediate tensors allocated by linear layers create large GPU memory spikes and significantly increase peak GPU memory usage of prefilling, while hybrid prefilling forwards the non-attention layers chunk-by-chunk and thus reduces the peak GPU memory usage.*

As shown in Figure 3a, we see that there are periodic spikes during the prefilling of the request. These spikes correspond to the intermediate tensors allocated to store the input and output of MLP module (a sequence of linear layers) inside Llama models. These tensors are large, as they contain 28672 floating numbers per token, 14× larger than the KV cache size of one layer. To illustrate why the intermediate tensors are large, we visualize the forward pass of the MLP module in the Llama-3.1-8B model in Figure 4, along with each intermediate tensor's shape and GPU memory occupation in bfloat16 precision.

We note that this trend—intermediate tensors allocated by linear layers are much larger than the KV cache size of one layer—also applies to existing LLM models. This is because existing LLM models choose to lower the size of KV cache (in order to increase the decoding throughput by enlarging the batch size, and also to handle longer LLM requests) without sacrificing the number of parameters in the LLM model. As a result, the LLM models have to inflate the tensor in MLP module, so that one can incorporate sufficient parameters into the linear layers.

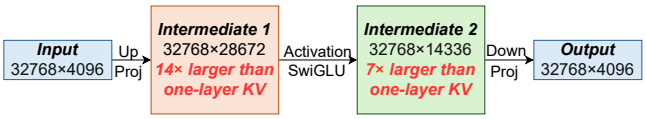


Figure 4. *Illustrating the tensor sizes of the MLP module in Llama-3.1-8B with bfloat16 precision. The size of intermediate tensors is much larger than the size of one-layer KV cache.*

4.2 Controlling intermediate tensors via hybrid prefilling

To control the size of intermediate tensors, we propose a new technique: *hybrid prefilling*, where we prefill the non-attention layers chunk-by-chunk and prefill the attention layers normally. Such hybrid prefilling will not change the LLM inference results, as the non-attention layers in LLMs are linear layers, and thus each chunk can be calculated separately from the other.

To show why hybrid prefilling reduces the GPU memory footprint of non-attention layers, we contrast traditional prefilling and hybrid prefilling in Figure 3. We see that, for traditional prefilling, the GPU memory trace contains high peaks because it stores the intermediate tensors for all input tokens, while in hybrid prefilling, we only store the intermediate tensor for just one chunk at any given point of time and thus the peak GPU memory usage is significantly reduced compared to traditional prefilling.

4.3 Implement hybrid prefilling via `torch.compile`

We employ `torch.compile` to implement hybrid prefilling. `torch.compile` is a new feature in PyTorch that allows the developer to change computation graph of one model without touching the model inference code.

We implement hybrid prefilling on top of the computation graph compiled by `torch.compile`. Concretely, we group the consecutive linear operations into a large virtual layer. Then, we forward through such large virtual layer chunk-by-chunk, and concatenate the output tensors of each chunk at the end.

We clarify some detailed optimizations to the hybrid prefilling process:

- *Output preallocation:* The above process requires concatenating the output tensor chunks to a large tensor, which doubles the GPU memory footprint of output tensor. To avoid this, we preallocate the output tensor before the forward pass using shape information inferred from the

computational graph, and directly write the output tensor for each chunk into this pre-allocated output tensor.

- *In-place computation*: We reuse the GPU memory of the input tensor to store the output tensor if they have the same shape. This is because the relative position between output chunk i and the output tensor is exactly the same as those between input chunk and the input tensor.

We evaluate the effectiveness of these optimizations in §7.

5 Enabling prefix caching

In this section, we illustrate how we enable prefix caching in PrefillOnly while still preserving the benefit of discarding the KV caches, and how PrefillOnly improves the scheduling algorithm in the context of prefix caching.

5.1 Suffix KV cache discarding

To simultaneously enable prefix caching and allow PrefillOnly to increase MIL by dropping KV caches, we propose suffix KV cache discarding, where PrefillOnly maximally preserves the KV cache of prefix tokens in GPU and discards the KV cache of suffix tokens.

We note that hybrid prefilling is the enabler of this technique, as hybrid prefilling only prefills each request within one LLM inference, allowing one to potentially discard part of the KV cache without worrying about slowing down the inference.

Also, our implementation does not change the hardware kernels as we reuse the abstractions created by sliding window attention in vLLM [2] to implement suffix KV cache discarding.

5.2 Comparing with parallelization

Dropping KV caches improves throughput when no prefix cache reusing: Compared to other approaches that increase MIL, PrefillOnly can process requests at the highest throughput as it does not parallelize the LLM inference or chunk the attention operation. As a result, when there is no prefix cache reusing, one should use PrefillOnly to maximally improve the throughput.

Parallelization can lower latency under high-speed inter-GPU interconnection: However, tensor parallelization can improve the latency of the LLM serving system under low QPS at the cost of extra communication. As a result, if the GPUs are interconnected with high-speed hardware like NVLink, one can use tensor parallelization to maximally reduce the latency.

6 Scheduling in the context of prefix caching

In this section, we discuss the scheduling algorithm of PrefillOnly in the prefix caching context. We first discuss why PrefillOnly does not choose to batch prefill-only requests. Then, we characterize two requirements for the scheduling algorithm in the context of prefix caching, and propose continuous JCT calibration as a general component for JCT-aware scheduling to meet the two requirements.

6.1 Why not batching prefill-only requests

In traditional LLM workloads, to maximize the output generation throughput (*i.e.* decoding throughput), the inference engine needs to maintain a large batch size by continuously batching new requests to the LLM engine and removing finalized requests from the batch. This technique significantly improves throughput, because the output generation phase is bounded by GPU memory accessing bandwidth, and batching 2× requests only marginally increases the total GPU memory needs to be accessed (as the major part is the LLM model weights) and thus marginally increases the runtime, but doubles the generation throughput.

However, as shown in §2.4, the inference in prefill-only workload is typically GPU computation-bound, and thus, batching does not significantly improve the throughput of processing Thus, batching prefill-only requests increases the average latency compared to processing the requests one by one, and does not improve the throughput. As a result, PrefillOnly chooses to schedule the requests one by one instead of batching them.

6.2 Limitation of traditional JCT-based scheduling

We observe that, traditional JCT-based scheduling algorithm has a low prefix cache hit rate in the context of prefix caching, resulting in high latency and low throughput.

This is because the JCT changes over time when prefix caching is enabled: the JCT of one request reduces when the prefix cache related to this request enters the LLM engine, and increases when this prefix cache is evicted. As a result, this approach fails to timely prioritize those requests that can hit the prefix cache, and when the LLM engine executes these requests, the prefix cache might already be evicted.

This is because traditional JCT-based scheduling algorithms make decisions based on the JCT when the request arrives, and thus fail to timely prioritize the requests when the LLM engine receives prefix cache related to these requests, and deprioritize the requests when the corresponding prefix cache of these requests is evicted.

Example: To further understand why the traditional JCT-based scheduling algorithm has low prefix cache hit rates. We provide an illustrative example. In this example, we assume that four requests (A, B, C, D) come into the LLM inference engine altogether, with request length A < C < B < D. Further, we assume A and D share the same prefix, and so do B and C. Also, we assume that the prefix cache space is limited and thus can only store the state of one request.

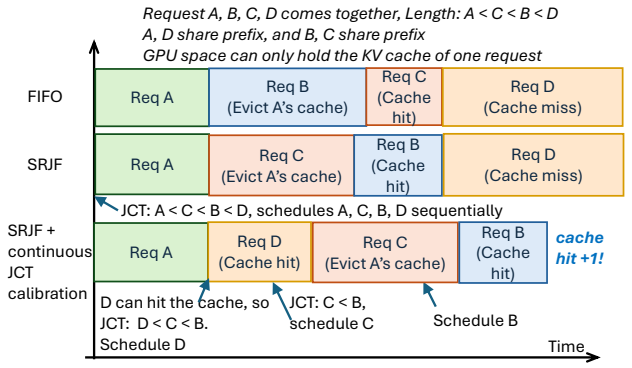


Figure 5. *Contrasting first-in-first-out (FIFO) scheduling, shortest-remaining-job-first (SRJF) scheduling and PrefillOnly's SRJF scheduling with continuous JCT calibration. The scheduling of PrefillOnly yields one more cache hit, achieving lower average latency.*

In this case, traditional shortest remaining job first scheduling will schedule the job based on its JCT, which is proportional to the request length. As a result, it will schedule the request in the order of A, C, B, D. In this case, request B can hit the prefix cache of request C, and thus request B can be accelerated. However, request D that could have hit the prefix cache of A, cannot be accelerated (as request C evicted the state of A), leading to high inference latency.

6.3 Continuous JCT calibration

Algorithm 1 SRJF with continuous JCT calibration

Input: Waiting queue Q of requests
Output: Request to schedule in the next step
$r_{\text{shortest}} \leftarrow$ None
$score_{\text{min}} \leftarrow \infty$
for each request $r \in Q$ **do**
    $n_{\text{input}} \leftarrow$ the number of input tokens in r
    $n_{\text{cached}} \leftarrow$ the number of tokens in r that hits prefix cache
    $T_{\text{queue}} \leftarrow$ the queuing time of r
    $score \leftarrow \text{get\_jct}(n_{\text{input}}, n_{\text{cached}}) - \lambda \cdot T_{queue}$
    if $score < score_{\text{min}}$ **then**
        $score_{\text{min}} \leftarrow jct$
        $r_{\text{shortest}} \leftarrow r$
    end if
end for
Schedule r_{shortest}

To improve the prefix cache hit rate of JCT-based scheduling algorithm, we propose PrefillOnly employs *continuous JCT calibrates*, where PrefillOnly calibrates the JCT of waiting requests every time before scheduling.

Such calibration significantly improves the prefix cache hit rate, as it allows the scheduling algorithm to timely prioritize those requests that can hit the prefix cache (as they typically have much lower JCT than other requests), which makes these requests much more likely to hit the prefix cache.

Example: To further illustrate why continuous JCT calibration helps JCT-based scheduling algorithms, like SRJF, improve latency, we give an illustrative example using the same setup as §6.2. The first scheduled job will be A. When scheduling the next job, the scheduler will perform JCT calibration, where it finds out that the JCT of D is significantly lowered as D can hit the prefix cache of A. As a result, the second scheduled job will be D. Then, the scheduler calibrates the JCT of B and C again, and their JCT remains unchanged. As a result, the scheduler then schedules C as it is shorter (and thus has lower JCT). After that, the scheduler will then schedule request B. In this case, the total number of cache hits is 2, where the total number of cache hits is 1 in both FIFO scheduling and naive SRJF scheduling. Thus, SRJF with continuous calibration achieves lower latency and higher throughput.

Calibration details: As shown in Algorithm 1, PrefillOnly calibrates the JCT of a given request r by calculating the number of input tokens n_{input} and the number of tokens that hits the prefix cache n_{cached}, and generate the JCT of this request by calling $jct(n_{\text{input}}, n_{\text{cached}})$, where we obtain jct by profiling how the JCT varies with respect to different pairs of n_{input} and n_{cached} that covers the maximum input length with the granularity of 1000 tokens, and trains a small linear model using linear regression.

Empirically, however, we found that the number of tokens that do not hit the prefix cache (*i.e.,* $n_{\text{input}} - n_{\text{cached}}$) is a good proxy of JCT: we measure that, on 1× A100 the Pearson correlation coefficient between the actual JCT and the number of cache miss tokens $n_{\text{input}} - n_{\text{cached}}$ is 0.987 on Qwen 32B with FP8 quantization (where 1 means perfectly correlated). As a result, PrefillOnly uses this JCT proxy by default.

Preventing starvation: In order to prevent starvation, PrefillOnly will reduce the JCT by $\lambda \cdot T_{req}$, where T_{req} is the queuing time of the request and λ is a hyperparameter: increasing the λ result in better worst-case latency at the trade of worse average latency.

We summarize the scheduling algorithm in Algorithm 1.

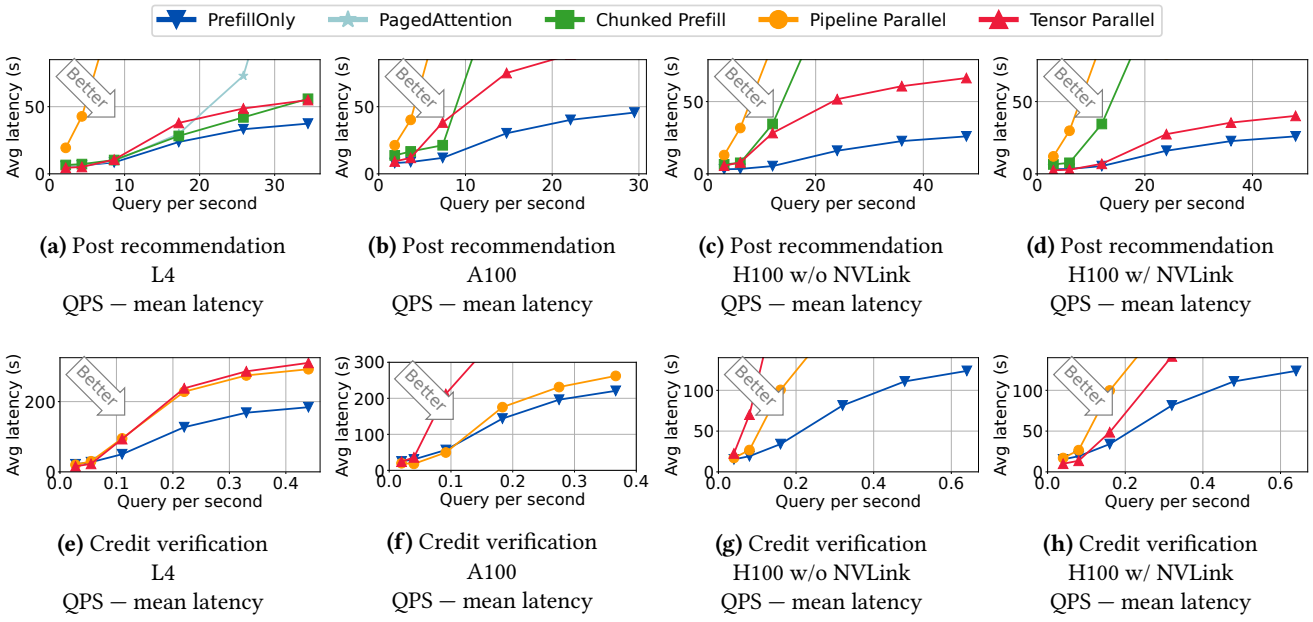
7 Evaluation

Our evaluation shows that:

- PrefillOnly handles 1.4 – 4.0× larger query-per-second without inflating the average latency and P99 latency compared to baselines.
- PrefillOnly expands the maximum request length by upto 5× without requiring parallelizing the LLM inference.

Dataset	Why evaluating this dataset	# of users	User profile length	Post length	# of requests per user	Total # of tokens
Post recommendation	Evaluate the ability of PrefillOnly under frequent prefix cache reuse	20	11, 000 tokens — 17, 000 tokens	150 tokens	50 (Each request corresponds to one post)	14,000,000
Credit verification	Evaluate the ability of PrefillOnly under long input length	60	40, 000 tokens — 60, 000 tokens	N/A	1	3,000,000

Table 1. *Summarizing the dataset used in the evaluation of PrefillOnly and baselines.*



(a) Post recommendation L4 QPS — mean latency

(b) Post recommendation A100 QPS — mean latency

(c) Post recommendation H100 w/o NVLink QPS — mean latency

(d) Post recommendation H100 w/ NVLink QPS — mean latency

(e) Credit verification L4 QPS — mean latency

(f) Credit verification A100 QPS — mean latency

(g) Credit verification H100 w/o NVLink QPS — mean latency

(h) Credit verification H100 w/ NVLink QPS — mean latency

Figure 6. *QPS — mean latency trade-off of PrefillOnly and baselines on four different hardware setups and two applications. PrefillOnly significantly reduces the latency when the QPS is high and only has higher QPS than tensor parallel baseline when QPS is low. Though tensor parallelism sometimes have lower latency than PrefillOnly under low QPS, it has much lower throughput than PrefillOnly due to extra communication cost and thus scales much worse than PrefillOnly in high QPS.*

7.1 Evaluation setup

In this subsection, we introduce the dataset, GPUs, LLM models, evaluation metrics, and baselines in our evaluation.

Existing LLM datasets mainly focus on evaluating the LLM accuracy instead of the performance of the LLM engine. As a result, in our evaluation, we use two simulated datasets, covering two tasks: post recommendation on a social media platform and credit verification for a bank application.

We summarize our evaluation dataset in Table 1.

Post recommendation dataset: In this dataset, we aim to evaluate the benefit of PrefillOnly under a short context scenario, where the major benefit of PrefillOnly comes from its scheduling. Concretely, we simulate a post recommendation scenario, where we recommend 10 out of 50 posts for a given user, based on the browsing history of this user. The key attributes from the perspective of LLM engine performance are the following parameters:

- *Post length:* To get a rough estimation of the post length, in terms of number of tokens, we take X as an example, and measured that the number of tokens for a short X post is less than 150 tokens. As a result, we use 150 tokens as the post length.
- *Number of posts to be recommended per user*: We set the number of posts to be recommended per user as 50. We assume that these 50 posts are given by the underlying recommendation systems using heuristics like embedding-based similarity search.
- *User profile length*: As for the user profile length, we focus on the click history of one user, where the user has already been engaged with the social media four times a week, and for four weeks. We assume that each time the user only clicks on five or six posts. As a result, the total length of the profile history is roughly 11,000 to 17,000 tokens. As a result, we use a normal distribution

Config	Max input length (measured by number of tokens)		
	L4	A100	H100
Paged Attention	24,000 WL1: ✓, WL2: ×	11,000 WL1: ×, WL2: ×	15,000 WL1: ×, WL2: ×
Chunked Prefill	46,000 WL1: ✓, WL2: ×	17,000 WL1: ✓, WL2: ×	25,000 WL1: ✓, WL2: ×
Pipeline Parallel	72,000 WL1: ✓, WL2: ✓	38,000 WL1: ✓, WL2: ✓	183,000 WL1: ✓, WL2: ✓
Tensor Parallel	195,000 WL1: ✓, WL2: ✓	77,000 WL1: ✓, WL2: ✓	238,000 WL1: ✓, WL2: ✓
PrefillOnly (ours)	130,000 WL1: ✓, WL2: ✓	87,000 WL1: ✓, WL2: ✓	97,000 WL1: ✓, WL2: ✓

Table 2. *Evaluating the max input length that PrefillOnly and baselines can handle under various hardware setups. WL1 indicates the post recommendation workload, and WL2 indicates the credit verification workload. × means that the max input length is insufficient to run corresponding workload.*

Scenario	GPU Type	LLM Model
Low-end GPU	2x NVIDIA L4 PCIe (24 GB)	meta-llama/ Llama-3.1-8B
Middle-end GPU	2x NVIDA A100 PCIe (40 GB)	RedHatAI/ DeepSeek-R1-Distill-Qwen-32B-FP8-dynamic
High-end GPU	2x NVIDIA H100 PCIe (80 GB)	Infermatic/ Llama-3.3-70B-Instruct-FP8-Dynamic
High-end GPU w/ NVLink	2x NVIDIA H100 NVLink (80 GB)	Infermatic/ Llama-3.3-70B-Instruct-FP8-Dynamic

Table 3. *The hardware and the corresponding LLM.*

to simulate the user profile length, with a mean as 14,000 and a standard deviation of 3,000.

- *Number of users*: We evaluated 20 users in total.

Credit verification dataset: In this dataset, our goal is to simulate a credit verification scenario, where we verify the credit of one user based on the credit history of one user. We measure that the length of credit history for one month is about 4,000 to 6,000 tokens. We simulate ten months of credit history, resulting in a credit history length from 40,000 to 60,000 for each user. We consider 60 users in total.

Request arrival pattern: We assume that the user arrival pattern is a Poisson process. We further vary the rate in the Poisson process to vary the query-per-second.

Hardware and LLM setup: We summarize the hardware and the LLM setup in Table 3.

PrefillOnly: We implement PrefillOnly based on state-of-the-art LLM serving engine vLLM [2], with 4.6k lines of Python code to implement the core techniques of PrefillOnly. We set the fairness parameter $\lambda = 500$ by default.

Baselines: We pick four baselines, where two of them parallelize the LLM inference (tensor parallel and pipeline parallel) and the other two do not parallelize the inference (PagedAttention [22] and chunked prefill [3]):

- *Tensor parallel*. In this baseline, we parallelize the inference onto 2 GPUs with the degree of tensor parallelism equal to 2 using the existing implementation available in production-grade inference engine vLLM [2].
- *Pipeline parallel*. In this baseline, we parallelize the inference onto 2 GPUs with the degree of pipeline parallelism equal to 2. We also use the implementation in vLLM [2].
- *PagedAttention* [22]. This baseline manages the KV caches using a page table to minimize fragmentation and employs first-come-first-serve scheduling.
- *Chunked prefill* [3]. This baseline processes the LLM input chunk-by-chunk to allow handling longer requests.

Note that we enable prefix caching for both PrefillOnly and all these baselines. Also, some baselines cannot handle some workloads as their maximum input length is too short, we show this in Table 2

Routing: We note that, for PrefillOnly and non-parallelization-based baselines, in order to utilize multiple GPUs, we launch multiple instances of LLM inference engines, one on each GPU, and then perform *user-id-based routing*, where we route the request from the same user to the same instance, and decide which user should be assigned to which instance in a round-robin manner.

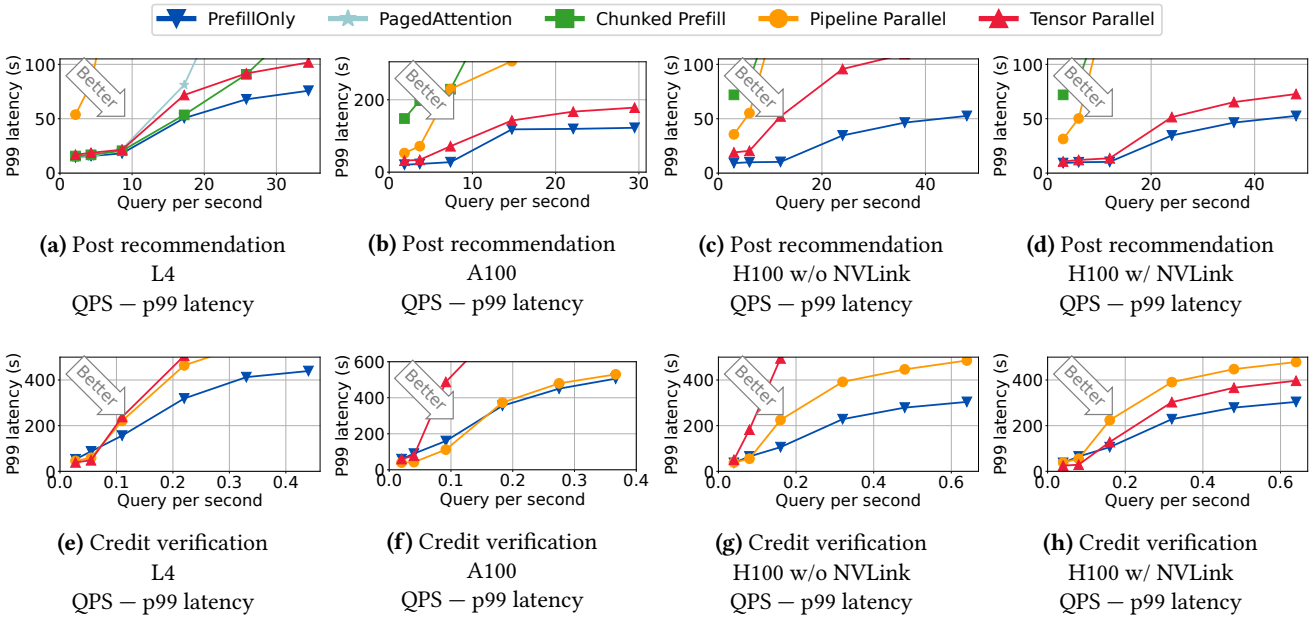
7.2 Evaluation results

QPS–latency trade-off: We show the trade-off between query per second (QPS) and latency (mean latency and p99 latency) across three different hardware setups and two different applications in Figure 6. In this figure, we determine the evaluation QPS by running PrefillOnly with all requests in the dataset coming at once, and then obtain the throughput (requests per second) of PrefillOnly in this situation, where we denote this value as x. We then evaluate QPS $1/4x, 1/2x, x, 2x, 3x, 4x$. This approach allows us to show the full spectrum performance of PrefillOnly.

In Figure 6, we see that PrefillOnly always achieves lowest latency when the QPS is high, indicating that the throughput of PrefillOnly is significantly higher than the baselines. However, in low QPS the latency of PrefillOnly can be higher than parallelization-based baselines (as baselines use multiple GPUs to serve one request but PrefillOnly just uses one).

We also show in Figure 7 that PrefillOnly achieves better P99 latency than baselines, indicating that the JCT-based allocation of PrefillOnly does not hurt P99 latency after applying the fairness twist mentioned in §6.3.

Source of improvement: We analyze the source of improvement of PrefillOnly on two datasets.



(a) Post recommendation L4 QPS — p99 latency

(b) Post recommendation A100 QPS — p99 latency

(c) Post recommendation H100 w/o NVLink QPS — p99 latency

(d) Post recommendation H100 w/ NVLink QPS — p99 latency

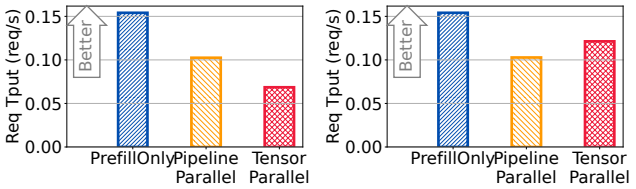
(e) Credit verification L4 QPS — p99 latency

(f) Credit verification A100 QPS — p99 latency

(g) Credit verification H100 w/o NVLink QPS — p99 latency

(h) Credit verification H100 w/ NVLink QPS — p99 latency

Figure 7. *QPS — P99 latency trade-off of PrefillOnly and baselines on three different hardware setups and two applications.*



(a) Throughput w/o NVLink

(b) Throughput w/ NVLink

Figure 8. *Contrasting the throughput of PrefillOnly and baselines on credit verification workload under 2× H100. Though NVLink significantly accelerates the communication and thus enhances the throughput of communication-intensive parallelization like tensor parallel, PrefillOnly still has the highest throughput as it does not spend extra communication to parallelize the inference.*

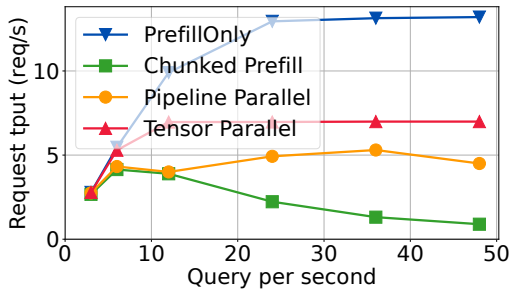


Figure 9. *Illustrating the throughput of PrefillOnly and the baselines in post post-recommendation dataset under 2× H100 without NVLink. PrefillOnly has better improvement as it can maintain high throughput under high query-per-second, while the query per second of chunked prefill baseline drops because of prefix cache throttling. Parallelization-based baselines parallelize the prefix cache across GPUs and thus have sufficient prefix cache space to avoid throttling, but they have lower throughput because of extra communication and synchronization cost.*

- *Post recommendation* To understand the source of improvement of PrefillOnly, we show how the throughput of PrefillOnly and baselines varies with respect to query per second in Figure 9. We can see that the query per second of chunked prefill baseline drops because the prefix cache throttles under high QPS, and PrefillOnly avoids such throttling by using continuous JCT calibration to identify and prioritize requests that can hit the prefix cache. Parallelization-based baselines parallelize the prefix caches across GPUs and thus have sufficient prefix cache space to avoid throttling, but they have lower throughput because of extra communication and synchronization cost.

- *Credit verification*: The main reason PrefillOnly performs better than other baselines under high QPS is because PrefillOnly can handle long context without parallelizing LLM inference, where a parallelization-based solution will have GPU idle time due to expensive all-reduce communication in the tensor parallel baseline, or the pipeline bubbles in the pipeline parallel baseline. To visualize the effect of GPU communication, we contrast the throughput of PrefillOnly and parallelization solutions between

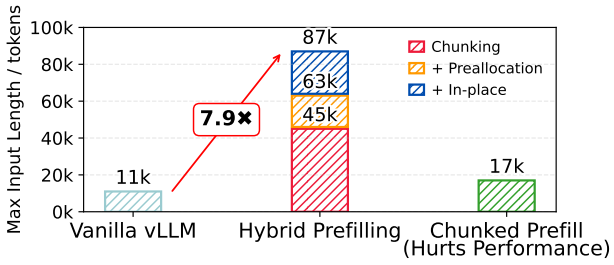


Figure 10. *Hybrid prefilling improves the MIL by* 7.9× *without hurting the throughput, measured on a Qwen-2.5-32B model with fp8 quantization on an A100 GPU.*

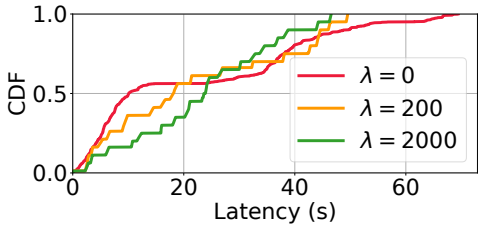


Figure 11. *The CDF of request latency of PrefillOnly, under different value of fairness parameter* λ*. Higher* λ *result in better P99 latency, at the cost of inflating the average latency.*

using NVLink and not using NVLink in Figure 8. We can see that, though having NVLink can significantly improve the throughput of the tensor parallel baseline due to much faster all-reduce communication, PrefillOnly still has better throughput as it does not spend extra communication to parallelize the inference.

Hybrid prefilling largely improves MIL: PrefillOnly can improve the maximum input length of LLM to upto 5× compared to the non-parallelization-based baseline (as shown in Table 2) without reducing throughput. Further, to show that hybrid prefilling can effectively enlarge the maximum input length (MIL), we plot how our individual techniques contribute to MIL. From Figure 10, we can see that compared to the chunked prefill baseline, PrefillOnly can improve the maximum context length of a Qwen-2.5-32B model in fp8 precision on an A100 GPU by more than 8.7×. We note that, though chunked prefill can also improve the MIL, it sacrifices both latency and throughput since it chunks the input and thus reduces GPU kernel efficiency.

Varying the fairness parameter λ**:** We vary the CDF of request latency of PrefillOnly, under different values of fairness parameter λ in Figure 11. Higher λ results in better P99 latency, at the cost of inflating the average latency

8 Related Work

LLM inference engines: After HuggingFace transformers standardized the way people access open-source LLMs, people started to engineer LLM inference engines that can serve LLMs with high throughput and low latency. Orca [50] introduces continuous batching that schedules LLM inference in token granularity instead of request granularity, allowing the LLM to significantly enlarge the batch size and improve the decoding throughput. vLLM [22] further lowers the GPU memory management granularity from request granularity to token granularity, thus minimizing GPU memory fragmentation, which leads to larger batch sizes and thus larger decoding throughput. DistServe [57] disaggregates prefill and decode across GPUs which significantly improves the goodput under TTFT and TPOT SLO constraints.

However, these inference engines are suboptimal for prefill-only workload, as the prefill-only request does not require decoding, so fully storing the KV caches unnecessarily enlarges the GPU footprint and does not accelerate this request. Also, the JCT of prefill-only requests is deterministic, which enables one to improve the scheduling by predicting JCT.

LLM caching systems: Beyond simple prefix caching [19, 29, 56], recent literature started to explore KV cache compression [17, 20, 31, 32, 54] to reduce KV cache size, and KV cache blending [18, 48, 55] to reuse non-prefix KV cache. CacheGen [31] compresses and streams KV states to reduce loading time for long prompts. CacheBlend [48] merges KV caches from multiple documents, selectively recomputing minimal attention states to support efficient RAG-style inference. PrefillOnly is compatible with these works, as PrefillOnly does not change the KV caches.

Building applications via LLM: Recent studies have started to leverage LLM to build a wide range of applications, including agentic systems [35, 46, 47, 52], recommendation systems [12, 42, 44] and more. This paper focuses on LLMs for discriminative applications, which typically requires the LLM to generate single output token for long input requests.

Traditional deep learning systems: Prefill-only workload resembles characteristics that are similar to traditional deep learning systems, where traditional deep learning systems also eliminate unnecessary tensors generated during inference [4] and leverage JCT-aware scheduling to improve performance [15, 36]. However, the GPU memory footprint of traditional systems may come from non-linear layers like convolution layers, and the JCT is roughly a constant [6, 9, 10, 26, 28]. PrefillOnly instead embraces LLM-specific properties that the GPU memory footprint mainly comes from linear layers, and the JCT is prefix-caching-sensitive, and develops optimizations on top of these properties.

9 Discussion

Offloading the KV caches to CPU: Current implementation of PrefillOnly performs suffix KV caches discarding,

which prevents future requests to potentially reuse the computation of the discarded part. This limitation can be alleviated by offloading the KV caches to CPU instead via solutions like LMCache [33]. We leave this extension to future work.

Prefill-decode disaggregation: Besides prefill-only workload, we found that PrefillOnly can be used on the prefill node in prefill decode disaggregation scenario, as the workload on the prefill node is also prefill-only. We leave this extension to future work.

Latency-centric optimizations: PrefillOnly focuses on exploiting the optimization opportunities in prefill-only workload that improves throughput. However, we also find that latency can be further optimized for prefill-only requests. For example, instead of paging the GPU memory [23], in prefill-only workload we can directly allocate continuous GPU buffers to accelerate GPU computation.

10 Conclusion

Besides generative LLM workloads, we observe that LLMs are increasingly being used in traditional discriminative tasks such as recommendation, credit verification, and data labeling. A new workload — the prefill-only workload — emerges under this trend, where LLMs generate only a single output token per request. Existing LLM engines perform suboptimally on such workloads, as their designs are specialized for multi-token generation. In this paper, we propose PrefillOnly, the first LLM inference engine tailored specifically for prefill-only workloads. PrefillOnly improves throughput for long requests by enabling one to offload or discard the KV caches during inference without slowing down the inference, thus avoiding parallelizing the KV caches when handling long requests, which hurts throughput. Furthermore, PrefillOnly introduces a scheduling algorithm based on continuous calibration of job completion time, reducing latency and improving cache hit rate. We evaluate PrefillOnly on 4 different hardware setups, 3 different models, and 2 types of workloads and show that PrefillOnly can handle up to 4× higher queries per second without increasing latency, paving the way for empowering day-to-day user applications through recommendation-based LLM pipelines.

References

[1] character.ai | personalized ai for every moment of your day. https://character.ai/. (Accessed on 09/07/2024).

[2] Extensions in arc: How to import, add, & open – arc help center. [Online; accessed 2025-04-17].

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

[5] Anysphere. The ai code editor. https://cursor.com/, 2025.

[6] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[7] Guozhu Dong and Huan Liu. *Feature engineering for machine learning and data analytics*. CRC press, 2018.

[8] Yixin Dong, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*, 2024.

[9] Kuntai Du, Yuhan Liu, Yitian Hao, Qizheng Zhang, Haodong Wang, Yuyang Huang, Ganesh Ananthanarayanan, and Junchen Jiang. Oneadapt: Fast adaptation for deep learning applications via backpropagation. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 158–176, 2023.

[10] Kuntai Du, Qizheng Zhang, Anton Arapin, Haodong Wang, Zhengxu Xia, and Junchen Jiang. Accmpeg: Optimizing video encoding for accurate video analytics. *Proceedings of Machine Learning and Systems*, 4:450–466, 2022.

[11] Duanyu Feng, Yongfu Dai, Jimin Huang, Yifang Zhang, Qianqian Xie, Weiguang Han, Zhengyu Chen, Alejandro Lopez-Lira, and Hao Wang. Empowering many, biasing a few: Generalist credit scoring through large language models. *arXiv preprint arXiv:2310.00566*, 2023.

[12] Hamed Firooz, Maziar Sanjabi, Adrian Englhardt, Aman Gupta, Ben Levine, Dre Olgiati, Gungor Polatkan, Iuliia Melnychuk, Karthik Ramgopal, Kirill Talanine, et al. 360brew: A decoder-only foundation model for personalized ranking and recommendation. *arXiv preprint arXiv:2501.16450*, 2025.

[13] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.

[14] GitHub. Github copilot - write code faster. https://copilot.github.com/, 2025.

[15] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[16] Xingwei He, Zhenghao Lin, Yeyun Gong, Alex Jin, Hang Zhang, Chen Lin, Jian Jiao, Siu Ming Yiu, Nan Duan, Weizhu Chen, et al. Annollm: Making large language models to be better crowdsourced annotators. *arXiv preprint arXiv:2303.16854*, 2023.

[17] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

[18] Junhao Hu, Wenrui Huang, Haoyi Wang, Weidong Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. Epic: Efficient position-independent context caching for serving large language models, 2024.

[19] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457*, 2024.

[20] Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. Gear: An efficient kv cache compression recipe for near-lossless generative inference of llm, 2024.

[21] Max Kuhn, Kjell Johnson, Max Kuhn, and Kjell Johnson. Over-fitting and model tuning. *Applied predictive modeling*, pages 61–92, 2013.

[22] 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*, pages 611–626, 2023.

[23] 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*, pages 611–626, 2023.

[24] Maxime Labonne and Sean Moran. Spam-t5: Benchmarking large language models for few-shot email spam detection. *arXiv preprint arXiv:2304.01238*, 2023.

[25] Xiaochong Lan, Yiming Cheng, Li Sheng, Chen Gao, and Yong Li. Depression detection on social media with large language models. *arXiv preprint arXiv:2403.10750*, 2024.

[26] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 359–376, 2020.

[27] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[28] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th annual international conference on mobile computing and networking*, pages 1–16, 2019.

[29] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E. Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing llm queries in relational workloads, 2024.

[30] Xiao-Yang Liu, Guoxuan Wang, Hongyang Yang, and Daochen Zha. Fingpt: Democratizing internet-scale data for financial large language models. *arXiv preprint arXiv:2307.10485*, 2023.

[31] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 38–56, New York, NY, USA, 2024. Association for Computing Machinery.

[32] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

[33] LMCache. Github - lmcache/lmcache: Redis for llms. [Online; accessed 2025-04-17].

[34] OpenAI. Chatgpt: Conversational language model. https://chat.openai.com, 2025.

[35] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

[36] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[37] Perlexity AI. Perplexity is a free ai search engine. https://www.perplexity.ai/, 2025.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[39] Yubo Shu, Haonan Zhang, Hansu Gu, Peng Zhang, Tun Lu, Dongsheng Li, and Ning Gu. Rah! recsys–assistant–human: A human-centered recommendation framework with llm agents. *IEEE Transactions on Computational Social Systems*, 2024.

[40] Adam Sobieszek and Tadeusz Price. Playing games with ais: the limits of gpt-3 and similar large language models. *Minds and Machines*, 32(2):341–364, 2022.

[41] Guijin Son, Hanearl Jung, Moonjeong Hahm, Keonju Na, and Sol Jin. Beyond classification: Financial reasoning in state-of-the-art language models. *arXiv preprint arXiv:2305.01505*, 2023.

[42] Yan Wang, Zhixuan Chu, Xin Ouyang, Simeng Wang, Hongyan Hao, Yue Shen, Jinjie Gu, Siqiao Xue, James Y Zhang, Qing Cui, et al. Enhancing recommender systems with large language model reasoning graphs. *arXiv preprint arXiv:2308.10835*, 2023.

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[44] Likang Wu, Zhi Zheng, Zhaopeng Qiu, Hao Wang, Hongchao Gu, Tingjia Shen, Chuan Qin, Chen Zhu, Hengshu Zhu, Qi Liu, et al. A survey on large language models for recommendation. *World Wide Web*, 27(5):60, 2024.

[45] Yufei Xia, Lingyun He, Yinguo Li, Nana Liu, and Yanlin Ding. Predicting loan default in peer-to-peer lending using narrative data. *Journal of Forecasting*, 39(2):260–280, 2020.

[46] Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*, 2023.

[47] Yi Yang, Yitong Ma, Hao Feng, Yiming Cheng, and Zhu Han. Minimizing hallucinations and communication costs: Adversarial debate and voting mechanisms in llm-based multi-agents. *Applied Sciences*, 15(7):3676, 2025.

[48] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 94–109, New York, NY, USA, 2025. Association for Computing Machinery.

[49] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[50] 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)*, pages 521–538, 2022.

[51] Carlos Vladimiro González Zelaya. Towards explaining the effects of data preprocessing on machine learning. In *2019 IEEE 35th international conference on data engineering (ICDE)*, pages 2086–2090. IEEE, 2019.

[52] Qizheng Zhang, Ali Imran, Enkeleda Bardhi, Tushar Swamy, Nathan Zhang, Muhammad Shahbaz, and Kunle Olukotun. Caravan: practical online learning of in-network ml models with labeling agents. In *Proceedings of the 3rd Workshop on Practical Adoption Challenges of ML for Systems*, pages 17–20, 2024.

[53] Ruoyu Zhang, Yanzeng Li, Yongliang Ma, Ming Zhou, and Lei Zou. Llmaaa: Making large language models as active annotators. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13088–13103, 2023.

[54] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[55] Shiju Zhao, Junhao Hu, Rongxiao Huang, Jiaqi Zheng, and Guihai Chen. Mpic: Position-independent multimodal context caching system for efficient mllm serving, 2025.

[56] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024.

[57] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.