CONTINUUM: EFFICIENT AND ROBUST MULTI-TURN LLM AGENT SCHEDULING WITH KV CACHE TIME-TO-LIVE

Hanchen Li[1], **Qiuyang Mang**[1], **Runyuan He**[1], **Qizheng Zhang**[2], **Huanzhi Mao**[1], **Xiaokun Chen**[3], **Hangrui Zhou**[4], **Alvin Cheung**[1], **Joseph Gonzalez**[1], **Ion Stoica**[1]

[1]UC Berkeley [2]Stanford University [3]Tensormesh [4]Tsinghua University

ABSTRACT

KV cache management is essential for efficient LLM inference. To maximize utilization, existing inference engines evict finished requests' KV cache if new requests are waiting. This policy breaks for agentic workloads, which interleave LLM calls with tools, introducing pauses that prevent effective KV reuse across turns. Since some tool calls have much shorter durations than human response multi-turn chatbot, it would be promising to retain the KV cache in during these tools. However, there are many challenges. First, we need to consider both the potential cost of recomputation or reloading (if CPU offloading enabled) and the increasing queueing delays after eviction from GPU. Second, due to the internal variance of tool call durations, we need the method to remain robust under limited predictability of tool call durations.

We present Continuum, a serving system to optimize job completion time for multi-turn agent workloads by introducing time-to-live mechanism for KV cache retaining. For LLM request that generates a tool call, Continuum selectively pins the KV cache in GPU memory with a time-to-live value determined by considering both the reload cost and ordering preserve benefit of retaining KV cache. Moreover, when the TTL expires, the KV cache can be automatically evicted to free up GPU memory, providing robust performance under edge cases. When combined with program-level first-come-first-serve, Continuum preserves multi-turn continuity, and reduces delay for complex agentic workflows. Our evaluation on real-world agentic workloads (SWE-Bench and BFCL) with Llama-3.1 8B/70B shows that Continuum significantly improves the average job completion times and its improvement scales with turn number increase.

1 INTRODUCTION

KV Cache management is key to inference engine performance, impacting both the input processing (prefill) and output generation (decoding) stages (Kwon et al., 2023; Zheng et al., 2024; Cheng et al., 2025). A critical component of KV cache management is the eviction policy. Ideally, the system should avoid evicting tokens that will be referenced in the immediate future. Similar to traditional caching systems, existing inference engines assumes that KV cache are less important once decoding is finished. This means that they will be discarded if other new requests in the waiting queue to maximize utilization. We refer to this type of policy **end-of-turn eviction**.

While end-of-turn eviction works well for multi-turn chat applications, it can significantly degrade the performance of modern agentic workloads, particularly those involving tool calling. These agentic applications have become increasingly popular across domains such as software engineering (Yang et al., 2024), computer use (Anthropic, 2024), and scientific research (Ren et al., 2025). These workloads characteristically interleave (a) inference steps to derive the next action, and (b) execution steps where the action is used to call an external tool. The output of the tool is subsequently appended to the request context, and a new inference step is initiated in the inference engine. The core issue arises when the request's KV cache is evicted when the agent transforms from inference step to tool call. If the KV cache was evicted for the finished request, the engine must recompute the prefix (prefill) or reload from CPU (if CPU offloading is enabled (Cheng et al., 2025)) when the tool

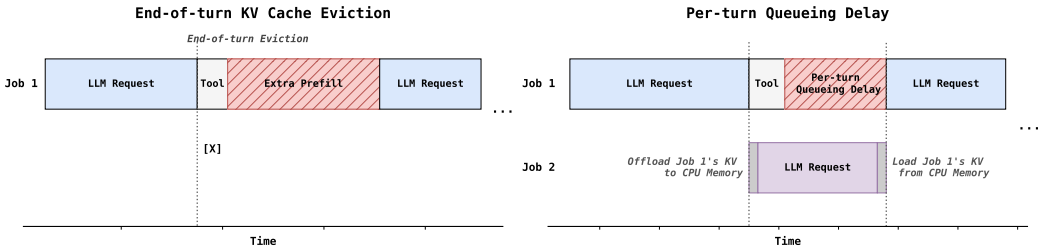


Figure 1: *Two main failures of previous agent serving work, where red blocks represent unnecessary overhead: even with CPU offloading, agents still suffer from queueing delay after eviction.*

execution completes and the next inference step begins. This repetitive prefill introduces substantial delays and reduces overall system throughput. More importantly, even when CPU offloading is enabled to greatly reuse KV cache, eviction causes **per-turn queueing delay** for the program. When the next inference step after a tool call has its KV cache evicted from GPU memory, it will also have to wait in the waiting queue for other requests to free up GPU memory before starting inference. This per-turn queueing delay can accumulate and result in increasing delay for each agentic program as illustrated in Figure 1. Since this delay is not measurable by offline profiling, we need to design a model to include this into our consideration. Moreover, since tool calls can be inherently variable, we need to set a maximum KV cache retainment time to prevent infinitely long waiting. However, if this time expires just before the tool call, the previous waiting will be wasted. Thus, we need to carefully set the KV cache retainment time to adapt to the workload.

Previous work fails to address these challenges. InferCept (Abhyankar et al., 2024) does not consider per-turn queueing delay and only statically preserves KV cache assuming fully predicatable tool calls. Autellix (Luo et al., 2025) uses end-of-turn eviction and ignores the importance of KV cache retention in multi-turn agent scheduling. Pie (Gim et al., 2025) only give interfaces but not policies for KV cache retention. Ayo (Tan et al., 2025), Alto (Santhanam et al., 2024), Parrot (Lin et al., 2024) assumes static workflow and do not apply to dynamic agents.

In this paper, we present Continuum, an agent serving system that utilizes KV cache time-to-live technique to improve job completion time for multi-turn agent workloads. Inspired by previous caching papers, Continuum introduces a KV cache time-to-live (TTL) mechanism to retain KV cache inside GPU after request finishes to over-ride original end-of-turn evictions. For each LLM request that generates a tool call during the inference step, Continuum models both the prefill/reload cost and the per-turn queueing delay reduction brought retaining KV cache. After obtaining the benefits of a potential hit based on the above two factors and tool call distributions, Continuum compares this with the cost of occupying GPU memory space during the TTL time to decide how long the KV cache can stay in GPU memory before being automatically evicted. This allows the next request to immediately resume if the tool call returns within the TTL window to save prefill and queueing delay. When the tool call prediction is inaccurate and the tool call takes longer than expected, Continuum can correct the mistake robustly by evicting the KV cache after the TTL expires, preventing severe memory pressure or deadlocks. Furthermore, Continuum combines the TTL mechanism with program-level first-come-first-serve scheduling. This enforces better request ordering and simplifies scheduling for complex agentic workflows.

To evaluate Continuum's performance, we conduct extensive experiment on real agentic workloads in function calling (Mao et al., 2024b) and coding agents (Lieret et al., 2025). Across three hardware and model setups, we show that Continuum reduces delay by 1.12x to 3.66x and improves throughput by 1.10x to 3.22x on multi-turn agentic workloads. Moreover, we evaluated Continuum on Company A's[1] internal testbed and show it can reduce delay for real SWE-agent workloads by up to 8.18x.

[1]Inference startup anonymized for double-blind

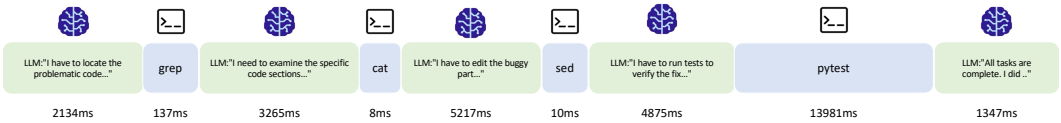


Figure 2: *Illustrative example of a SWE-Agent. The agent resolves a software engineering bug step by step with tool calls in the middle. These tool calls have different durations and breaks the continuity of the LLM inference.*

Dataset	No. of Turns	Tool Time(ms)	Token Per Program
SWE-Bench	(10.9, 2.1)	(925, 3,550)	(70,126, 19,732)
BFCL v4	(6.3, 2.3)	(1,923, 2,133)	(93,256, 68,687)

Table 1: *Statistics from two collected datasets. Numbers are in format of (mean, standard deviation).*

2 Background

2.1 ReAct Paradigm for Agents

Most modern agentic workloads mainly follow the paradigm of the *ReAct*-agent loop (Yao et al., 2022), alternating between a reasoning step where LLM tries to understand contexts and output thoughts, and perform external actions via tool calls. An emerging trend is that many agentic applications and evaluation suites scale this loop into *long-horizon, multi-turn* iterations, repeatedly interleaving thought, tool call and context update across dozens (or even hundreds) of turns, e.g. the τ-bench benchmark for tool-agent-user interaction (Yao et al., 2024), the MINT benchmark for multi-turn tool-augmented interaction (Wang et al., 2023), and AgentBench for evaluating LLMs in multi-turn, open-ended decision-making and tool-use scenarios (Liu et al., 2023).

This interleaved reasoning and tool-call paradigm features a few unique advantages in the context of agentic workloads. Thus, the ReAct framework has gradually become the de facto standard of agentic workloads. Coding agents including Claude Code (Anthropic / Claude, 2025) and Cursor (Cursor, 2025) almost all adopt the ReAct workflow. Frameworks such as LangChain (LangChain, 2025) and LangGraph (LangGraph, 2025) implement ReAct-style agents and graph-based state machines for explicit state to enable more developers build ReAct style agents.

More importantly, the most recent leading open-source models like GPT-OSS (Agarwal et al., 2025) and Kimi-K2 (kim, 2025) take things further, where it bakes the tool call ability into the base model, and allows for ReAct-style tool calls inside the reasoning chain.

2.2 Agentic Traces

We collect and analyze 100 traces from mini-swe-agent (Lieret et al., 2025) running SWE-Bench (Jimenez et al., 2024) and 100 traces from BFCL V4 Agentic Web Search (Mao et al., 2025), both running GPT-5 as the base model. Figure 2 presents an illustrative shortened example trace from SWE-Bench, demonstrating how the agent solves a software engineering task step by step.

The takeway is three-fold. First, there are many turns for these novel agentic programs. This increase in turn numbers adds additional scheduling difficulty. Second, the tool call times have varying time distribution, but many are short. Although the request will be considered finished after these short tool calls are generated, the next request will arrive soon after the tool call execution completes, reusing the KV cache.

2.3 Limitations of Existing Methods

Previous works fail to handle this emerging complex workload due to three main reasons:

Fixed Workflow: One line of work focused on improving scheduling of agentic workflows with **pre-defined, static** computation graphs. Teola (Tan et al., 2025) decomposes applications into primitive-level dataflow graphs, exposing fine-grained dependencies between LLM and non-LLM components. Alto (Santhanam et al., 2024) focuses on streaming and pipelined execution across distributed components. Parrot (Lin et al., 2024) exposes application-level context to LLM services

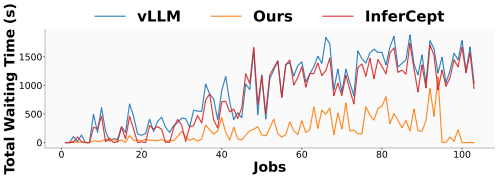
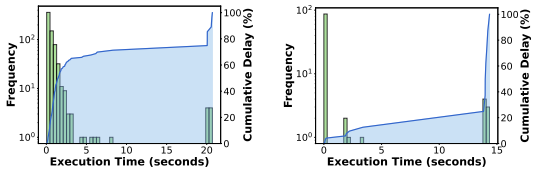


Figure 3: *Per-turn queueing causes high waiting time in baselines even with CPU offloading.*



(a) BFCL: fetch_url (b) SWE-Bench: cd

Figure 4: *Execution-time skew: the slowest 10% of calls dominate total delay.*

through Semantic Variables, enabling the engine to infer data dependencies across consecutive LLM requests. One shared limitation of Teola, Parrot, and Alto is that they all assume static or deterministically defined DAGs and **could not work with dynamic agent workloads** like ReAct-styled ones whose dependency graphs evolve at runtime. This limits the capabilities of these works to optimize for the wide variety of agents in practice (Anysphere, 2024; Lieret et al., 2025; Yan et al., 2024).

No Consideration for Tool Calls: Autellix (Luo et al., 2025) introduces Program-Level Attained Service (PLAS) scheduling that prioritizes requests with less cumulative service time of the agentic program. Tempo (Zhang et al., 2025) proposes a scheduler to satisfy the SLOs when facing different types of requests (chat, agent, reasoning), while our focus is particularly on agentic workloads with many-turn and variable tool calls. These work fail to consider the unique characteristics of tool calls in agentic workloads, such as their variable durations and the impact on KV cache management. This oversight can lead to suboptimal scheduling decisions and increased latency, demonstrated as the **end-of-turn KV eviction** in Figure 1.

Insufficient KV Cache Retainment Strategies: Some previous work observed the challenge of KV cache reuse for agent workloads. InferCept (Abhyankar et al., 2024) introduces a "preserve" operation that pins the KV cache between tool calls. However, their policy overlooks the multi-turn nature of requests. When KV cache is evicted between turn, this will cause additional queueing time per turn for the program when they come back. We demonstrate the performance degradation brought by this **per turn queueing** for multi-turn scheduling in Figure 3. We profile the total eviction overhead experienced by each request for vanilla vLLM and the InferCept algorithm. The x-axis represents each agentic program in order of arrival time, while the y-axis denotes the total bubble time for each agentic job — the total idle period a request experiences in the waiting queue before execution. Even with InferCept's KV retainment, bubbles still persist and causes delay increase despite its throughput improvement over vLLM. Moreover, their preserve operation is fixed and could not adapt to tool use in real time. If the actual tool call time is much longer than predicted, blindly "preserving" the KV cache can cause significant inefficiency. However, as shown in Figure 4, many tool calls exhibit high variability in execution time.

Pie (Gim et al., 2025) introduces a programmable serving system that decomposes the generation loop into fine-grained handlers. While its optimizations can retain KV cache across tool calls with manual flag, it still lacks the ability to make principled, real-time decisions about KV cache retention, a s it provides no mechanism to adapt to dynamic tool-call latencies or multi-turn dependencies.

3 Continuum Scheduling Algorithm

Given the failure of previous work, we claim that an optimal KV cache retainment policy should include the following features:

- It should retain KV cache for requests that will reuse them soon after tool calls, minimizing prefill/loading overheads.
- It should consider the multi-turn continuity of agent programs, reducing waiting time and preserving program order.
- It should be robust to varying tool call latencies.

In order to achieve the robustness guarantee, we propose to borrow the idea of Time-to-live (TTL) from traditional systems: for each request's KV cache, we give a TTL value to define the maximum

Notation	Description	Notation	Description
τ	TTL	$\mathsf{MemUsage}(r)$	GPU memory occupied by r
$\mathcal{M}$	Avg. memory of seen requests	$\mathsf{CacheMissCost}(r)$	Cost of reloading r
$\mathsf{Prefill\text{-}Reload}(r)$	KV reconstruction time	$\mathsf{OutofOrderCost}(r)$	Out-of-order cost for r
η	Memoryfulness factor	$\mathcal{T}$	Average waiting time
$\mathcal{P}(\tau, f)$	Finish-within-TTL prob.		

Table 2: *Key notations in Continuum's cost model for a request r and tool-call f.*

duration for it to remain in GPU memory. This prevents long-running or failed tool calls from blocking GPU resources indefinitely while retaining KV cache.

3.1 Utility Model

To set an effective TTL value (in seconds) for pinning a request's KV cache, Continuum must choose the value that best balances the benefit of potential reuse against its cost. Both the benefit and the cost are measured in units of time, since they ultimately translate into changes in the total job completion latency across all programs. Mathematically, given a request r and a TTL value τ, Continuum estimates $\mathsf{Cost}(\tau, r)$ and $\mathsf{Benefit}(r)$ for pinning the KV cache of request r for τ. For simplicity, $\mathsf{Benefit}(r)$ assumes that the next request arrives within the TTL window. The case where TTL expires before the tool call returns is addressed in Sec. 3.2.

Cost Estimation. The cost of pinning a request's KV cache comes from the opportunity cost of occupying GPU memory that could otherwise be used to serve other requests:

$$\mathsf{Cost}(\tau, r) = \frac{\mathsf{MemUsage}(r)}{\mathcal{M}} \times \tau,$$

where $\mathsf{MemUsage}(r)$ is the amount of GPU memory used by the KV cache of request r, $\mathcal{M}$ is the average GPU memory footprint of active requests, and τ is the TTL value.

The ratio $\frac{\mathsf{MemUsage}(r)}{\mathcal{M}}$ represents how many average requests are blocked when r is pinned. In other words, if pinning r occupies the same memory as k requests, then pinning r adds τ latency to approximately k other requests. We assume that the waiting queue always contains enough requests for this blocking effect to occur when KV retainment is necessary.

Benefit Estimation. The benefit of pinning a request's KV cache is realized when the request is re-issued within the TTL period, allowing it to avoid the overhead of reloading or prefilling the KV cache from r's program while saving the per-turn queueing delay:

$$\mathsf{Benefit}(r) = \mathsf{CacheMissCost}(r) + \mathsf{OutofOrderCost}(r)$$

We use the sum of cost prevented as the benefit. $\mathsf{CacheMissCost}(r)$ measures the cost of reloading or prefilling the KV cache for request r and $\mathsf{OutofOrderCost}(r)$ measures the expected queueing delay for the request due to waiting for other requests to free GPU memory.

Similar to $\mathsf{Cost}(\tau, r)$, we can measure $\mathsf{CacheMissCost}(r)$ by (1) the context reconstruct overhead $\mathsf{Prefill\text{-}Reload}(r)$; and (2) the approximate number of requests will experience the additional latency overhead $\frac{\mathsf{MemUsage}(r)}{\mathcal{M}}$. The cost is formally defined as follows:

$$\mathsf{CacheMissCost}(r) = \frac{\mathsf{MemUsage}(r) \times \mathsf{Prefill\text{-}Reload}(r)}{\mathcal{M}}$$

$\mathsf{Prefill\text{-}Reload}(r)$ is the time cost for prefill or reloading depending on whether CPU offloading is turned on. This is based on a quick offline profiling described in Sec C.2.

Measuring the expected queuing delay: Innovatively, as we discussed in Sec. 2.3, immediately rescheduling the requests coming back from tool calls also helps reduce the latency caused by extra waiting time before scheduling the request even with KV cache offloading. Note that the queueing delay is closely tied to the memoryfulness of the workload, *i.e.,* whether the number of remaining steps reduce predictably as the program progresses.

Let N be the total number of requests in a program and k the number of requests that have already been served. We define the following *memoryfulness factor*

$$\eta = -\mathrm{Corr}(k, N - k)$$

We can see this factor models the degree of memoryfulness in the workload well: when the workload is fully memoryless, we have that k is independent to $N - k$, leading to $\eta = 0$. Conversely, when the workload is fully memoryful, *i.e.,* all programs have the same fixed number of requests, we have $\mathrm{Corr}(k, N - k) = \mathrm{Corr}(k, -k) = -1$, resulting in $\eta = 1$.

Now, we are ready to define the $\mathsf{OutofOrderCost}(r)$ based on the η above. When $\eta = 1$, the delay is exactly the waiting time when the program of r returns back to the waiting queue. To match this, we record the average waiting time per unit context size for the historical requests in this workload as $\frac{\mathcal{T}}{\mathcal{M}}$, where T is the average queueing delay for previous requests. In this case, the delay can be well measured by $\frac{\mathcal{T}}{\mathcal{M}} \times \mathsf{MemUsage}(r)$. Here, we consider $\mathsf{MemUsage}(r)$ since large-context requests are harder to schedule (they must wait for enough contiguous memory to be freed). For the general cases, we define the out-of-order cost as follows:

$$\mathsf{OutofOrderCost}(r) = \frac{\mathcal{T}}{\mathcal{M}} \times \mathsf{MemUsage}(r) \times \eta.$$

3.2 Setting the TTL Value

In this part, we describe how Continuum sets the TTL value for KV cache based on the cost-benefit model above and historical tool-call information. As in Algorithm 1 (line **??**), Continuum determines the optimal TTL value τ^* to maximize the expected net benefit of retaining the KV cache:

$$\tau^* = \mathrm{argmax}_\tau \; \mathcal{P}(\tau, f) \times \mathsf{Benefit}(r) - \mathsf{Cost}(\tau, r), \tag{1}$$

where $\mathcal{P}(\tau, f)$ estimates the probability that the tool call f completes within time τ. This formula captures the expected net benefit, in terms of total job latency, of retaining the KV cache of r for a duration of τ By eliminating the shared $\frac{\mathsf{MemUsage(r)}}{\mathcal{M}}$, the formula above can be transformed to

$$\mathrm{argmax}_\tau \; \mathcal{P}(\tau, f) \times \big(\mathcal{T} \cdot \eta + \mathsf{Prefill\text{-}Reload(r)}\big) - \tau, \tag{2}$$

indicating that we only need to additionally compute $\mathcal{T}$ and $\mathcal{P}(\tau, f)$ in our implementation. $\mathcal{T}$ can be estimated as the sliding window average for queueing delay experienced by requests who was evicted. Since we cannot fully predict the duration of the next tool call, we estimate $\mathcal{P}(\tau, f)$ using the empirical CDF derived from historical tool-call records $S[f]$. Specifically, we calculate it as the following:

$$\mathcal{P}(\tau, f) = \frac{1}{|S[f]|} \cdot \sum_{t \in S[f]} \mathbb{I}[t \le \tau]$$

, where $\mathbb{I}[\cdot]$ is the indicator function. Finally, we solve Equation equation 2 by enumerating all unique tool-call durations recorded in $S[f]$ as candidates (including $\tau = 0$) and selecting the one with the highest expected reward.

3.3 Putting it together

We present the full algorithm at Appendix 1. We implemented Continuum on top of vLLM with approximately 1,000 lines of Python. Our design introduces a `Tool-Call Handler` within the scheduler loop, which parses tool invocations and tracks historical latency statistics to compute optimal TTL values. We augmented the scheduler with primitives to pin and unpin requests, managing KV cache retention in a `pinned_requests` dictionary. `pinned_requests` maps program identifiers to their associated TTL values. While the TTL values has not expired, the scheduler prioritizes pinned requests in scheduling decisions. The scheduler loops cleans expired entries from `pinned_requests` and unpins their KV cache to free GPU memory at the beginning of each scheduling decision. To ensure system robustness, we integrated a deadlock prevention mechanism that preemptively unpins victims under memory pressure, and we calibrated the cost model using offline profiling of hardware-specific prefill and offloading latencies. Due to space limit, we leave detailed system implementations in Appendix C.

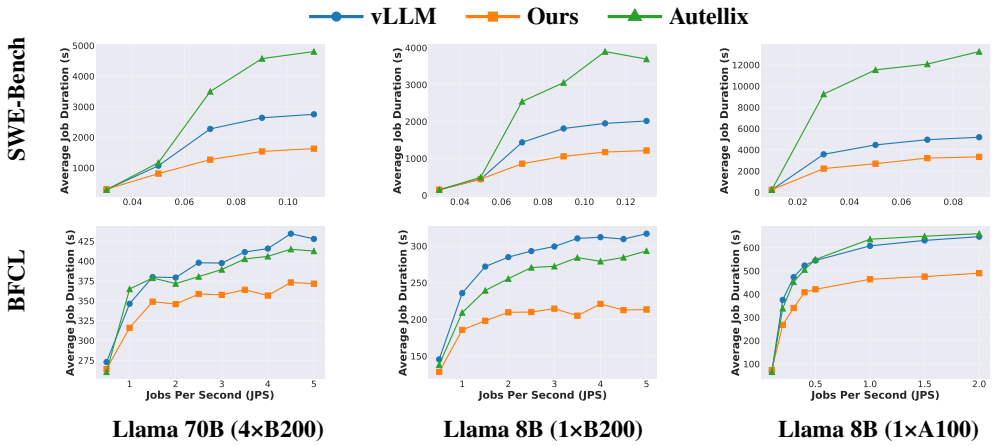


Figure 5: *Continuum outperforms against baseline schedulers across different model sizes, hardware configurations, and datasets.*

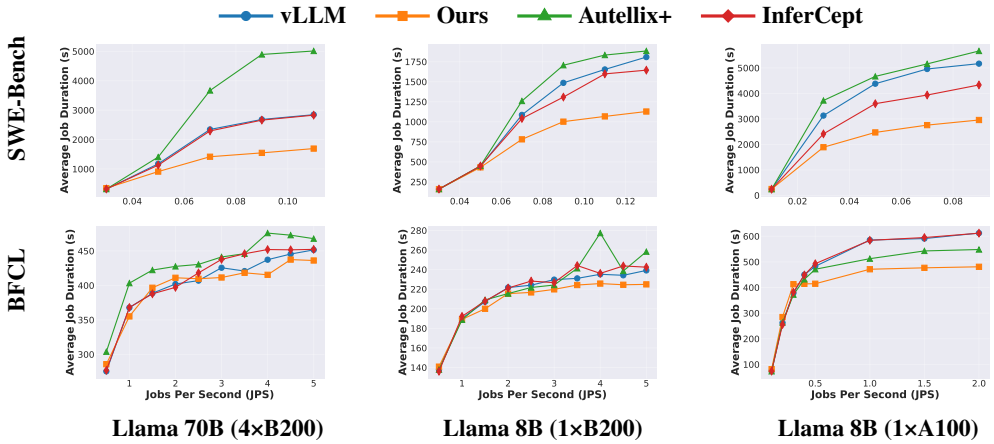


Figure 6: *Continuum achieves consistent improvement when DRAM offloading is enabled. It improves over systems with smart DRAM offloading logic like InferCept by considering tool-call and multi-turn together.*

4 Evaluation

4.1 Setup

Model and Hardware: We evaluate Continuum with llama-3.1-8B and llama-3.1-70B models. For 8B model, we use A100-SXM GPU from Runpod and B200 GPU from on-prem. For 70B model, we use H100 GPU from Company A and B200 from on-prem and set the tensor parallelism to 4.

Datasets: For results other than the real SWE-Bench experiments in Figure 7, we evaluate on two collected workloads running GPT-5 [2] and using poisson distribution for the arrival pattern of agents:

- SWE-Bench (Jimenez et al., 2023): We run mini-swe-agent (Lieret et al., 2025) on SWE-Bench. We keep requests within the context window.
- Berkeley Function Calling Leaderboard (Mao et al., 2025): We used BFCL V4 (Web Search category). This includes agents answering questions with web browsing tools. We scaled down the workload by 0.4 to fit at least 100 request in the context window of llama-3.1 (128k tokens).

Main Baselines:

[2]We use GPT-5 for the better model capabilities to ensure that the workflow generated are mostly correct. Base small models often fail to accomplish the task

- *Vanilla vLLM* We use the stable release of vllm 0.10.2 with default setting. Chunked-prefill is enabled with default chunk size 2048.
- *CPU DRAM offloading* We use vllm 0.10.2 with LMCache 0.3.7 (Cheng et al., 2025). For A100 GPUs, we set the DRAM size used in offloading to be 100GB; For B200 and H100 GPUs, we set the DRAM size used in offloading to be 200GB per GPU. We also apply this on baselines.
- *Autellix* We implemented the algorithm of PLAS from Autellix (Luo et al., 2025) on top of vllm. We extend Autellix to CPU offloading cases by enabling LMCache (Autellix+).
- *InferCept* We implemented the selectively preserve, swap, or evict algorithm of InferCept (Abhyankar et al., 2024) on top of vllm + lmcache. Since the CPU offloading in LMCache is non-blocking (better than original InferCept), we update the cost estimation accordingly.
- *Distributed Inference Solutions* We compare with other open-source solutions for real experiments including SGLang 0.5.5.post3 (sgl project) with native cache-aware routing and Dynamo 0.7.0.post1 (ai dynamo) configured with 1P1D for PD Disaggregation.

4.2 Experiment Results

Figure 5 and Figure 6 demonstrate the end-to-end improvement from Continuum. We show significant improvements in both average response time and throughput across both the BFCL and SWE-Bench workloads. For instance, with the Llama-3.1-8B model, Continuum achieves up to a 2x reduction in average response time compared to the vanilla vLLM baseline. The performance gains are consistent across different model sizes and hardware configurations, demonstrating the effectiveness of our approach in diverse scenarios. Although Autellix outperforms baselines in BFCL, it underperforms in SWE-Bench due to its false assumption that requests have longer expected finish time if they execute for longer.

Moreover, we observe that Continuum consistently outperforms CPU offloading baselines. On the other hand, PLAS's gain on CPU offloading diminished compared with baseline. This demonstrates Continuum's robust performance improvement on scheduling bubble reduction that is orthogonal to DRAM offloading techniques.

Real SWE-Agent in Distributed Setting: In order to fully evaluate Continuum's performance in real-world deployment scenarios at scale. We test Continuum running real SWE agent for 500 tasks in SWE-Bench-Verified in Company A's internal H100 testbed. We set up our agent client environment by adding a job distributor for the SWE-Bench platform that distributes agents in poisson distribution. We use a simple session aware routing for Continuum and compare against other distributed inference solutions. We measure the per-job finish time and collect the pass rate of each agent program for their generated results on SWE-bench after generation.

As demonstrated by Figure 7, Continuum consistently outperforms baselines in terms of average delay when pass rates are equal. Notice that Continuum actually has higher pass rate than baselines. This is due to SWE-Bench's time limit for environment dockers to prevent hanging. When the baseline's running time exceeds 15 minutes it will be preempted and treated as failure case. This proves Continuum's usability in real production settings.

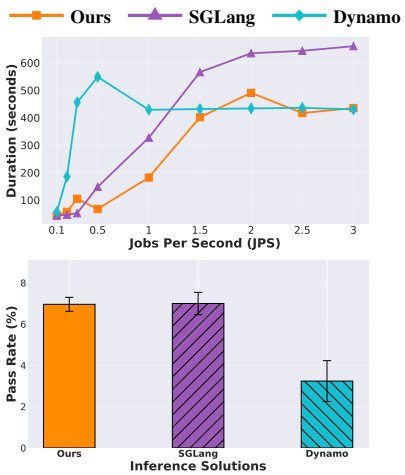


Figure 7: *Continuum improves delay under the pass rate for real SWE-agents in distributed settings.*

4.3 Sensitivity Analysis

Varying Inference Engine Configuration: In order to show that Continuum is robust to varying inference engine configurations, we evaluate Continuum with different configurations of the inference engine. In Figure 9, we set the job per second to be 0.13 and vary the maximum batch size to compare Continuum with different baselines. As we can see, Continuum's improvement remains stable across different batch sizes. Moreover, in Figure 9, we vary the number of chunk size from 256 to 4096. We observe similar improvements across different chunk sizes. This demonstrates the robustness of our approach to different inference engine configurations.

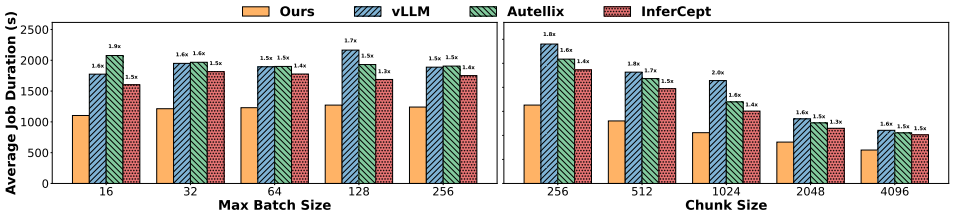


Figure 9: *Continuum improves delay across different max batch-size and chunk-size configurations.*

Robustness to Turn Numbers: Figure 8 evaluates our scheduler's robustness in multi-turn scenarios. We simulate more-turn scenarios on SWE-Bench by repeating the trace ($1\times$ to $5\times$) while inversely scaling the token lengths to emulate more turns but make total token fit within the context window. With a request rate of 0.13 JPS and 200 GB for DRAM offloading, the results show that the baseline methods degrade as the number of turns increases. This is because the increased number of turns leads to more tool calls and longer overall execution times, exacerbating the scheduling challenges faced by traditional methods. In contrast, our approach maintains stable, low-latency performance, demonstrating its effectiveness for complex, many-turn agentic interactions.

Ablation studies, turn number robustness studies, latency analysis are available in Appendix B.

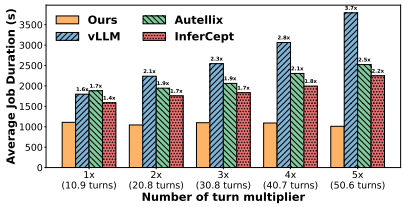


Figure 8: *Continuum shows higher improvement ratio as the number of turns increases.*

5 Related and Future Work

LLM Inference Systems: There have been many research papers on improving LLM inference. Serving engines including vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) achieves state of the art inference by adapting paged attention design and optimized kernels. Besides the wide range of kernel-level optimizations that improve GPU execution speed (Ye et al., 2025; Dao et al., 2022; Zhu et al., 2025), researchers have also proposed many optimizations on resource management: continuous batching (Yu et al., 2022), chunked prefill (Agrawal et al., 2024), skip-join multi-level scheduling (Wu et al., 2023). Many of them have been ported into the inference engine. Previous work have also explored efficient offloading to CPU DRAM and disks (Gao et al., 2024a; Xie, 2025; Cheng et al., 2025; Liu et al., 2024; Yao et al., 2025). For distributed inference, people have adopted session aware routing (Srivatsa et al., 2024; Lab & vLLM, 2025), KV-cache aware routing (Xia et al., 2025), and prefill-decode disaggregation (Zhong et al., 2024). Building upon these work, Continuum extends LLM inference into long-horizon multi-turn agentic workloads and improves resource management when resources are competed by different requests.

Generality Beyond ReAct-Style Agents: The current design and implementation of Continuum are optimized for ReAct-style, tool-interleaving agents where each LLM step returns a clear tool invocation followed by a gap before the next step. Continuum naturally extends to parallel tool calls since it still follows the sequential "reason $\rightarrow$ tool $\rightarrow$ reason" rhythm. Some emerging agent frameworks, however, could involve non-linear control flows: speculative branches, asynchronous multi-agent coordination, and context folding. Although such workloads are mostly experimental and yet to be tested in real production workloads, their inference pattern may violate the sequential flow and requires future change. In such settings, program-level FCFS ordering and per-program TTL pinning may not map cleanly to the underlying control flow and requires future updates. Extending Continuum to support branching traces, concurrent tools, or fine-grained internal state updates is an important direction for future work. More discussions are available in Appendix F.1.

6 Conclusion

Agentic workloads introduce new scheduling challenges for LLM serving systems due to frequent tool calls, highly variable inter-step delays, and the need to preserve multi-turn continuity. We present Continuum, a KV cache retention and scheduling system that balances both the benefit

of cache reuse and the cost of blocking GPU memory through a time-to-live mechanism. By integrating TTL-aware pinning with program-level FCFS, Continuum reduces unnecessary prefills, mitigates per-turn queueing delays, and robustly adapts to unpredictable tool-call latencies. Our implementation on top of vLLM shows consistent improvements in end-to-end job completion time and throughput across model sizes, hardware configurations, and real-world agent workloads. Continuum demonstrates that principled, tool-aware KV management is essential for efficient multi-turn agent serving. We hope it lays the groundwork for future systems to more deeply integrate agent-workload structure into LLM inference engines.

REFERENCES

Kimi k2 tech blog. https://kimi-k2.org/blog, 2025. Accessed: 2025-12-08.

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. Infercept: Efficient intercept support for augmented large language model inference. In *Forty-first International Conference on Machine Learning*, Vienna, Austria, July 2024.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

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)*, pp. 117–134, 2024.

ai dynamo. Dynamo. https://github.com/ai-dynamo/dynamo. Accessed: 2025-12-09.

Anthropic. Parallel tool calling transforms speed and performance. https://www.anthropic.com/engineering/built-multi-agent-research-system.

Anthropic. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku. https://www.anthropic.com/news/3-5-models-and-computer-use, 2024.

Anthropic / Claude. Claude code. https://claude.com/product/claude-code, 2025. Accessed: 2025-12-11.

Anysphere. Cursor: The ai code editor. https://cursor.com, 2024.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Zhipeng Chen, Kun Zhou, Beichen Zhang, Zheng Gong, Wayne Xin Zhao, and Ji-Rong Wen. Chatcot: Tool-augmented chain-of-thought reasoning on chat-based large language models. *arXiv preprint arXiv:2305.14323*, 2023.

Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240): 1–113, 2023.

Cursor. Agents — cursor. https://cursor.com/agents, 2025. Accessed: 2025-12-11.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. *arXiv preprint arXiv:2403.19708*, 52:20–38, 2024a.

Silin Gao, Jane Dwivedi-Yu, Ping Yu, Xiaoqing Ellen Tan, Ramakanth Pasunuru, Olga Golovneva, Koustuv Sinha, Asli Celikyilmaz, Antoine Bosselut, and Tianlu Wang. Efficient tool use with chain-of-abstraction reasoning. *arXiv preprint arXiv:2401.17464*, 2024b.

In Gim, Seung-seob Lee, and Lin Zhong. Asynchronous llm function calling. *arXiv preprint arXiv:2412.07017*, 2024.

In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. Pie: A programmable serving system for emerging llm applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, pp. 415–430, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400718700. doi: 10.1145/3731569.3764814. URL https://doi.org/10.1145/3731569.3764814.

Antonio A Ginart, Naveen Kodali, Jason Lee, Caiming Xiong, Silvio Savarese, and John Emmons. Asynchronous tool usage for real-time agents. *arXiv preprint arXiv:2410.21620*, 2024.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. *Advances in neural information processing systems*, 33: 1474–1487, 2020.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021a.

Albert Gu, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré. Combining recurrent, convolutional, and continuous-time models with linear state space layers. *Advances in neural information processing systems*, 34:572–585, 2021b.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Carlos E. Jimenez, John Yang, Kilian Lieret, Alex L. Zhang, and Ofir Press. Swe-bench: Can language models resolve real-world github issues? https://github.com/SWE-bench/SWE-bench, 2024.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. In *Forty-first International Conference on Machine Learning*, 2024.

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*, pp. 611–626, 2023.

LMCache Lab and vLLM. vllm production stack, 2025. URL https://docs.vllm.ai/projects/production-stack/en/latest/.

LangChain. React-style agents — langchain documentation. https://python.langchain.com/api_reference/langchain/agents/langchain.agents.react.base.ReActChain.html, 2025.

LangGraph. Stategraph and graph-based state machines — langgraph. https://langchain-ai.github.io/langgraph/concepts/agentic_concepts/, 2025.

Kilian Lieret, John Yang, Carlos E. Jimenez, Alexander Wettig, Shunyu Yao, Karthik Narasimhan, and Ofir Press. mini-swe-agent: The 100-line ai agent that resolves github issues on swe-bench. https://github.com/SWE-agent/mini-swe-agent, 2025.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of {LLM-based} applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 929–945, 2024.

Tim Lin. Overview of function calling in open-source models. https://medium.com/%40c22647809/overview-of-function-calling-in-open-source-models-cc23e9b13360, 2025.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56, 2024.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. Autellix: An efficient serving engine for llm agents as general programs. *arXiv preprint arXiv:2502.13965*, 2025.

Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Tianjun Zhang, and Shishir G. Patil. Bfcl v2 • live dataset. https://gorilla.cs.berkeley.edu/blogs/12_bfcl_v2_live.html, 2024a.

Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Jason Huang, Vishnu Suresh, Yixin Huang, Xiaowen Yu, Joseph E. Gonzalez, and Shishir G. Patil. Bfcl v3 • multi-turn & multi-step function calling evaluation. https://gorilla.cs.berkeley.edu/blogs/13_bfcl_v3_multi_turn.html, 2024b.

Huanzhi Mao, Raymond Tsao, Jingzhuo Zhou, Shishir G. Patil, and Joseph E. Gonzalez. Bfcl v4: Web search. https://gorilla.cs.berkeley.edu/blogs/15_bfcl_v4_web_search.html, 2025.

OpenAI. Parallel function calling in the openai api. https://community.openai.com/t/parallel-function-calling-vs-routing-to-functions-yourself/597886, 2024.

OpenAI. Introducing gpt-realtime and realtime api updates: Long-running function calls will no longer disrupt the flow of a session. https://openai.com/index/introducing-gpt-realtime/, 2025.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Qwen. Function calling — qwen documentation. https://qwen.readthedocs.io/en/latest/framework/function_call.html, 2024.

Shuo Ren, Pu Jian, Zhenjiang Ren, Chunlin Leng, Can Xie, and Jiajun Zhang. Towards scientific intelligence: A survey of llm-based scientific agents. *arXiv preprint arXiv:2503.24047*, 2025.

Keshav Santhanam, Deepti Raghavan, Muhammad Shahir Rahman, Thejas Venkatesh, Neha Kunjal, Pratiksha Thaker, Philip Levis, and Matei Zaharia. Alto: An efficient network orchestrator for compound ai systems. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pp. 117–125, 2024.

sgl project. sglang. https://github.com/sgl-project/sglang. Accessed: 2025-12-09.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for llm serving, 2024. URL https://arxiv.org/abs/2407.00023.

Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Towards end-to-end optimization of llm-based applications with ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 1302–1316, 2025.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

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. *arXiv preprint arXiv:2309.10691*, 2023.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

Wenxun Wu, Yuanyang Li, Guhan Chen, Linyue Wang, and Hongyang Chen. Tool-augmented policy optimization: Synergizing reasoning and adaptive tool use with reinforcement learning. *arXiv preprint arXiv:2510.07038*, 2025.

Tian Xia, Ziming Mao, Jamison Kerney, Ethan J. Jackson, Zhifei Li, Jiarong Xing, Scott Shenker, and Ion Stoica. Skywalker: A locality-aware cross-region load balancer for llm inference, 2025. URL https://arxiv.org/abs/2505.24095.

Zhiqiang Xie. Sglang hicache: Fast hierarchical kv caching with your favorite storage backends, 2025. URL https://lmsys.org/blog/2025-09-10-sglang-hicache/.

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Ion Stoica, Joseph E. Gonzalez, Tianjun Zhang, and Shishir G. Patil. Berkeley function-calling leaderboard. https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html, 2024.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

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*, pp. 94–109, 2025.

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*, 2022.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025. URL https://arxiv.org/abs/2501.01005.

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)*, pp. 521–538, 2022.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

Wei Zhang, Zhiyu Wu, Yi Mu, Banruo Liu, Myungjin Lee, and Fan Lai. Tempo: Application-aware llm serving with mixed slo requirements. *arXiv preprint arXiv:2504.20068*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37: 62557–62583, 2024.

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)*, pp. 193–210, 2024.

Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2025. URL https://arxiv.org/abs/2408.12757.

Algorithm 1 Continuum's Scheduling Algorithm

Require: Waiting queue Q; TTL map P (pinned programs and TTLs); historical tool-call records S where $S[f]$ stores records for tool f

function ONREQUESTARRIVE(r)
  $Q \leftarrow Q \cup \{r\}$
  $id \leftarrow$ program ID of r
  if id is a seen program **then**
    $(f, t) \leftarrow$ tool-call info from r
    $S[f] \leftarrow S[f] \cup \{t\}$
  end if
end function
function ONREQUESTFINISH(r)
  if r is the last request of its program **then**
    Free KV cache used by r
  else
    $f \leftarrow$ next tool after r
    $id \leftarrow$ program ID of r
    $P[id] \leftarrow \text{CALCTTL}(r, S[f])$
  end if
end function
function SCHEDULE
  while Q not empty **do**
    for all $id \in P$.keys **do**
      if current time $> P[id]$ and $id \notin Q$.programs **then**
        Free KV cache of id's last request
        Remove id from P
      end if
    end for
    $r \leftarrow \arg\max_{r' \in Q} \text{CALCPRIORITY}(r', P)$
    if r cannot fit in memory **then**
      break
    end if
    $Q \leftarrow Q \setminus \{r\}$
    Issue r to run
    $id \leftarrow$ program ID of r
    if $id \in P$.keys **then**
      Remove id from P
    end if
  end while
end function

A FULL ALGORITHM

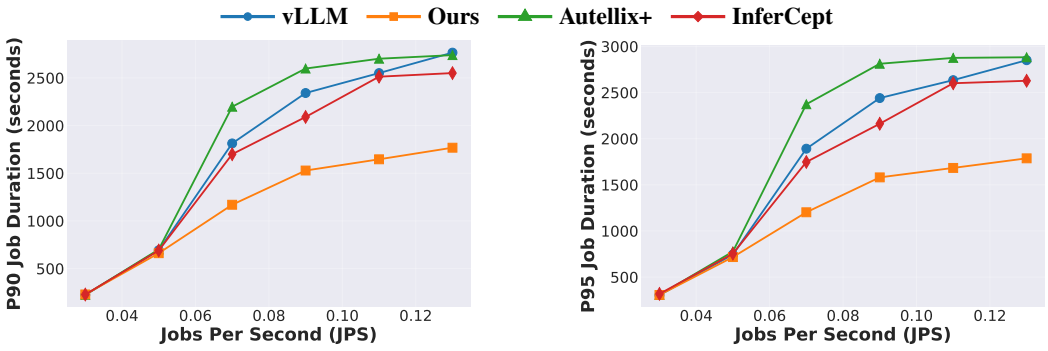
We present the full algorithm of Continuum in Algorithm 1.

B EXTENDED EXPERIMENTS

We show our improvements on P90 and P95 delay in Figure 10. Continuum achieves better P90 and P95 latency due to its ability to reduce the per-turn queueing delay compared with baselines, achieving similar gain as in average delay.

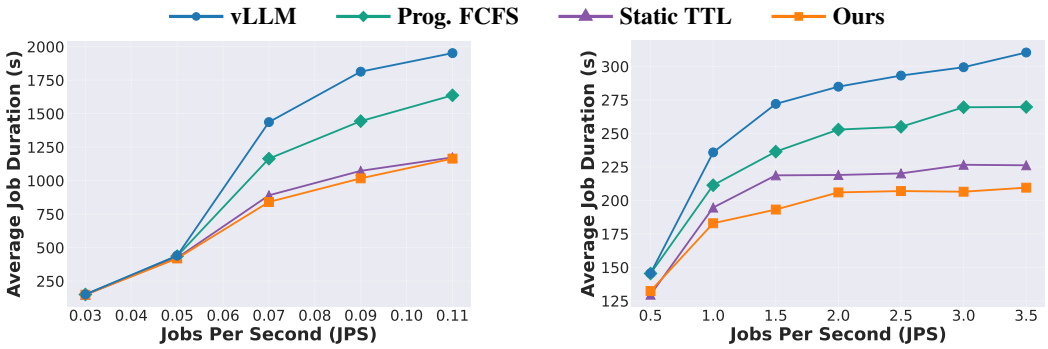
SSD Offloading: Similar to CPU offloading, SSD offloading offers bigger space but slower loading. We evaluate Continuum with extended SSD storage layer beyond CPU offloading using LMCache on SWE-bench workload with llama-8B on B200. As shown in Figure 12, Continuum consistently improves average delay compared with baselines when also utilizing disks of different sizes.

Ablation Study: We conduct an ablation study to analyze the impact of our cost modeling on Continuum's overall performance. In Figure 11, we compare Continuum with baselines that only applies part of the optimizaions. Program-Level FCFS changes the original request-level FCFS in vLLM into priority based on program arrival. Static TTL builds upon program-level FCFS to utilize



(a) P90 (b) P95

Figure 10: *Continuum achieves better P90 and P95 latency for running SWE Bench trace with Llama-8B model on B200 with CPU offloading turned on.*



(a) SWE-Bench (b) BFCL

Figure 11: *Contributions of individual ideas to Continuum. Program-level FCFS prioritizes requests with earlier program arrival. Static TTL uses fixed TTL threshold from cold start handling.*

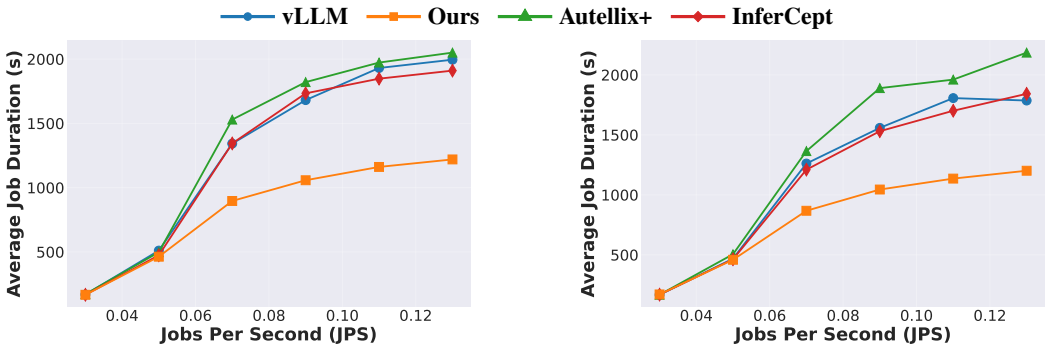
fixed TTL threshold estimated cold-start handling. As demonstrated, different ideas of Continuum gradually improves performance.

Scheduler Overhead: As shown in Table 3, our approach introduces a minor scheduling overhead compared to the baselines. However, this overhead is on the order of single-digit milliseconds, which is negligible compared to the GPU execution time for LLM inference. The significant end-to-end performance improvements from our scheduling strategy far outweigh this small increase in scheduling latency.

C Continuum System Design

In Continuum, our design goal is a modular architecture that requires minimal changes to the core inference-engine scheduler loop. On the client side, we attach a program identifier (`program_id`) to every inference request so the system can recognize multi-turn agent programs and reason about tool calls across steps.

Upon arrival at the serving engine, requests enter the existing scheduler loop. Continuum adds a thin Tool-Call Handler that is invoked on request arrival and completion. The handler parses tool calls from LLM outputs, tracks per-tool latency using observed inter-request intervals within the same `program_id`, and returns TTL to the scheduler. The scheduler uses this hint to pin the request's KV cache for potential reuse by the next step, and later unpins it either when the TTL value expires or when the program terminates.



(a) SSD = 400G (b) SSD = 800G

Figure 12: *Continuum reduces delay when we extend offloading device to SSDs beyond CPU offloading.*

System	No CPU Offload	CPU Offload
vLLM	0.95 ms	2.33 ms
Autellix	0.82 ms	2.18 ms
InferCept	N/A	2.25 ms
Ours	0.96 ms	2.30 ms

Table 3: *Continuum introduces minor scheduling latency overhead comparison under different DRAM offloading settings.*

C.1 Tool Call Handler

The tool call handler is a separate class invoked by the main scheduler after the arrival or at the finish of a request. This decoupled structure ensures that tool-related parsing and timing logic remain isolated from the core scheduling loop, ensuring extensibility for future parsers or tool-aware policies.

Identifying the Tool Call: When the scheduler completes request, it forwards the response to the tool-call handler, which determines whether the response includes a tool invocation. The handler parses the message according to the function call schema used by the serving API. In modern agentic APIs, LLM outputs frequently adopt a standardized tool call structure. For example, in the OpenAI schema:

```
{
  "id": "fc_0",
  "call_id": "call_0",
  "type": "function_call",
  "name": "get_weather",
  "arguments": {"location": "Paris"}
}
```

For this example schema, the handler checks each returned message block's `type`; if it indicates a function/tool call, the handler extracts the call's `name` and uses this as the tool call type. In SWE-Bench, it is guaranteed that each LLM's response containing a function call will include exactly one `bash` function call. We extract the string within the `bash` block and use the first word afterwards as the tool call name.

More function call format examples for different LLMs (Lin, 2025; Qwen, 2024) can be found in Appendix E. Continuum can be easily extended to these with a parser similar to Appendix D.

Recording the tool finish time: For each LLM request i in a program identified by a program ID p, the handler records a server-side completion timestamp $t^{p,i}_{\text{finish}}$ along with tool call name when

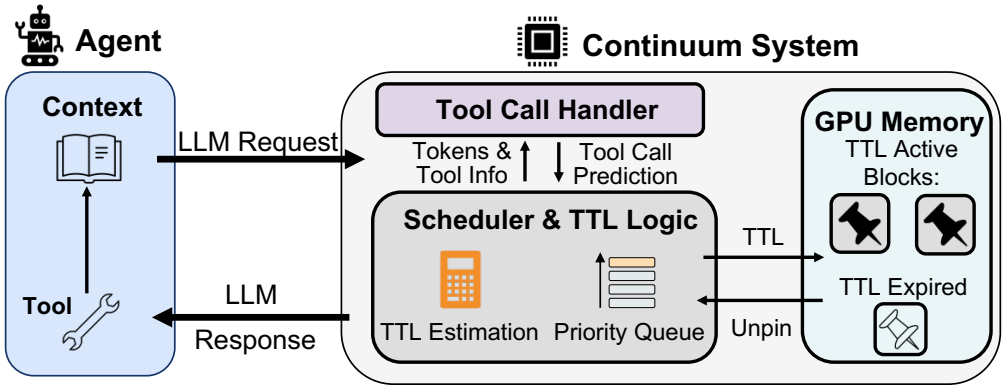


Figure 13: *System Overview of Continuum*

scheduler records a finished request with tool call output. When the next request $i+1$ with the same p arrives, we observe its server-side arrival timestamp $t^{p,i+1}_{\text{arrive}}$ and compute the inter-request interval $t^{p,i+1}_{\text{arrive}} - t^{p,i}_{\text{finish}}$. We record this interval as the execution time of the tool call this time to store for TTL computation in the future.

C.2 Efficient Pin with TTL in Scheduler

After the tool call handler gives the TTL value, the scheduler will need to execute the pin operation.

Request Pining: If the step is not signified to be the last step (ex. parsed to contain a tool call), the scheduler calls the tool-call handler to obtain the TTL value τ^* and, if not zero, invokes `pin_request(request, τ*)`. This records a pair of request and its expiration time `current_timestamp` $+ \tau^*$ in a dictionary `pinned_requests` and deliberately skips freeing the request's KV blocks. The `pinned_requests` will also be passed to the waiting queue to prioritize the scheduling of the next request in the same program.

Request Unpinning: At the beginning of every scheduling step, the scheduler runs `unpin_requests()`. It scans `pinned_requests` and unpins entries whose TTL have expired *and* whose `program_id` does not currently appear in the waiting queue. This prevents premature eviction when a follow-up request has already arrived at the inference engine but scheduler has not been able to schedule it. Additionally, when a program's last step finishes, the scheduler proactively unpins any remaining pins with the same `program_id`, as no KV cache reuse is expected in the near future.

Prevention of deadlocks: Pinned requests can accumulate and potential deadlock could occur when all the GPU memory is occupied by the pinned requests. Since the pinned requests would be preserved if the next request of the same program is still in the waiting queue, the entire scheduling loop could be stuck and no new requests can be scheduled to run due to the lack of space.

Thus, we need a mechanism to unpin the requests when the such a deadlock occurs. In Continuum, when the scheduling logic fails to schedule a new request to execute for the scheduling step, we iteratively selects victims from `pinned_requests` with the latest program arrival time to unpin and free the space until the first request can be scheduled to run.

The chosen request is removed from its queue position, its KV cache is freed, and it is re-queued as needed, ensuring that subsequent allocations can proceed to run. This prevents deadlock even when many pins are present.

Offline Profile: In order to predict the prefill time and reloading time ($\mathsf{Prefill\text{-}Reload}(r)$) based on context size as needed in Sec 3.1, we perform an offline profile on each hardware and model pair for online estimation. We profile for two purposese: **(1)** GPU-CPU bandwidth for CPU offloading cases. We measure by taking the average CPU offloading throughput. **(2)** Prefill vs context length curve for estimating prefill cost. We measure this by doing prefill for chunk sizes $\{1000, 2000, 4000, ...max\_context\_length\}$ and fit a quadratic curve on the data. Admittedly, there

could be some pages for the request remaining in GPU memory that does not need recomputation. But these remaining pages are usually small when memory is contended and we approximate by the full prefill time with little error. Profiling takes less than 10 minutes for each hardware model pair.

C.3 Implementation

We implemented Continuum on top of vLLM with about 1k lines of Python. Besides the above pinning operations added to the scheduler class, we use three functions from tool call handler in vLLM's original scheduler:

- `func_call_finish(tool, timestamp)`: When request finishes and parsed to contain tool call, this function informs tool call handler to record the tool call starting time.
- `update_tool_call_time(program_id, timestamp)`: When inference step is submitted, denotes a tool call finished if the previous inference request ended with tool call.
- `set_up_ttl(request, tool)`: Based on previous tool call information and the system setup, give best TTL value for the scheduler to this finished request.

D Tool Call Parser Implementation Example

We attach the implementation for the tool parser for mini-SWE-agent here.

```python
class ToolCallParser:
    """Parser for extracting function calls from LLM output.

    Uses the same parsing logic as mini-swe-agent to extract bash commands
    from markdown code blocks and identify the function call.

    This can be extended for other datasets with different parsing logic.
    """

    def parse(self, text: str) -> Optional[str]:
        """Parse LLM output and extract the function call name.

        Args:
            text: Output text from the LLM

        Returns:
            The function call name (e.g., "ls", "cd", "git"), or None if not found
        """
        # Same regex pattern as mini-swe-agent: r"```bash\s*\n(.*?)\n```"
        actions = re.findall(r"```bash\s*\n(.*?)\n```", text, re.DOTALL)

        if len(actions) == 1:
            bash_action = actions[0].strip()
            # Extract the first word (command) from the action
            words = bash_action.split()
            if words:
                return words[0]

        return None
```

Listing 1: Tool Call Parser Example

E More Function Call Examples

Under the hood, models differ in how they surface tool calls in their chat templates and generations. For instance, Llama-3 variants may emit a function-style string `func_name(param_1=val_1, param_2=val_2, ...)`, whereas Qwen-3 variants favor JSON such as `"name": "func_name", "arguments": ...`. Regardless of format,

serving engines (e.g., vLLM, SGLang) include model-specific, template-aware parsers that take in the generated long string, recover the function name and parameters, and normalize them into the OpenAI-style schema, enabling uniform downstream handling. Thus, if we are using the general function calling interface provided by the serving engines, we don't need to worry about model-specific parsing.

For other use cases where the application is not using the function calling interface, and instead ask the model to output structured bash command via the chat interface, it's also easy to parse out the function name and arguments.

For example, in SWE Bench, to extract the intended tool invocation, just locate the single bash code block, split the command string on && or ——, then parse each sub-command: the first token is the executable/function name (pytest, git, . . .) and the rest are its arguments.

```
pytest -q && git add -A && git commit -m "fix: handle None case in
    parser"
```

In Terminal Bench, this is even easier, as their structured format already handles the command splitting for us.

```
{
  "state_analysis": "The tests are failing with a NameError.",
  "explanation": "Open the file, fix the missing import and rerun
   tests.",
  "commands": [
    { "keystrokes": "vim src/app/main.py\n", "is_blocking": false,
    "timeout_sec": 2.0 },
    { "keystrokes": "pytest -q\n",         "is_blocking": true,  "
   timeout_sec": 30.0 }
  ],
  "is_task_complete": false
}
```

F Extended Discussions of Related Work

F.1 Novel Tool-Calling Styles

Thinking with tools: This pattern interleaves planning with execution: the model emits a structured intermediate plan, calls tools, integrates their feedback, and continues its chain of thought Agarwal et al. (2025); Gao et al. (2024b); Wu et al. (2025); Chen et al. (2023). In Continuum, once a tool call is emitted, the current request is considered complete; after the tool finishes, a follow-up request is enqueued with the updated context. Continuum can be extended to this scenario by implementing a tool parser as shown in in .

Parallel tool calls: When sub-tasks are independent (e.g., “”How is the weather in US and UK?”), issuing multiple tool calls in parallel can shorten turn latency Kim et al. (2024); Anthropic; OpenAI (2024); Mao et al. (2024a); Yan et al. (2024); Patil et al. (2025). By design, these calls are commutative: they may execute in any order, and their responses are appended to the context as they complete. Continuum can be extended through a function call predictor from client.

Asynchronous tools: Asynchronous tool calls make execution non-blocking: they enable token generation while the tools run in the background Gim et al. (2024); Ginart et al. (2024); OpenAI (2025). This is especially useful for breadth-first or tree-search behaviors (e.g., deep-research or browsing agents that fan out multiple probes concurrently). This workload suits Continuum well: because the model performs little active computation between awaits, KV-cache reuse is high as long as we avoid premature eviction. We leave support for parallel and asynchronous tool calls as future work.

F.2 Model Architecture

People have been proposing new LLM model architectures beyond the traditional decode-only transformers. Mix-of-Experts (MoE) Shazeer et al. (2017); Fedus et al. (2022); Chowdhery et al. (2023) introduces sparsity into the model by activating only a subset of parameters for each input token, enabling larger models with lower inference cost. Sliding window transformers Beltagy et al. (2020); Zaheer et al. (2020) limit the attention scope to a local window instead of the full context, reducing the memory footprint during inference. Hybrid Models combine full attention with more efficient attention mechanisms such as linear attention Choromanski et al. (2020); Katharopoulos et al. (2020), SSMs Gu & Dao (2024); Gu et al. (2021a; 2020; 2021b) or low-rank attention Wang et al. (2020) to reduce memory footprint and improve inference speed. These architectures alleviate the memory bottleneck during inference to achieve higher throughput, but they still suffer from the scheduling issues discussed, especially the scheduling bubbles due to different jobs' perpetual contention for GPU space.