Lookahead Context Engineering: Hiding Context Transformation Overhead for Efficient Long-Horizon Agent Serving

Zaifeng Pan, Qianxu Wang, Zhengding Hu, Chang Chen
Yue Guan, Steven Swanson, Yufei Ding
University of California, San Diego

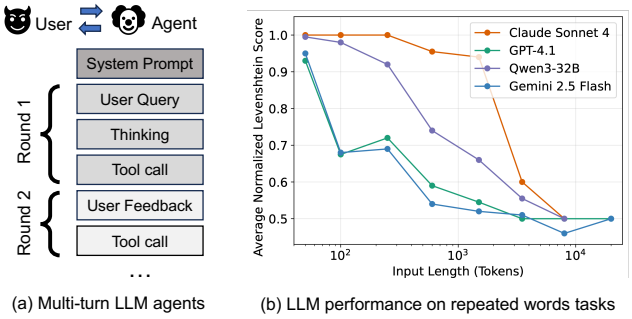
Abstract

LLM-based agents execute multi-turn workflows with continuously growing contexts, where LLM calls are interleaved with tool invocations and environment feedback. To maintain model quality, modern agent frameworks rely on context engineering strategies such as offloading, reduction, and isolation to control the context length. However, these strategies introduce significant context transformation overhead: each transformation invalidates existing KV caches and triggers re-prefill, leading to increased time-to-first-token (TTFT).

In this paper, we identify that context transformations are segment-decomposable, where the transformation of a prefix is independent of future tokens. This property enables transformations to be executed ahead of time. Based on this insight, we propose a lookahead programming model that allows agent frameworks to express context transformations as asynchronous operations without modifying their execution logic. The runtime proactively executes these transformations and prepares transformed KV caches in advance, enabling direct context replacement without blocking. We further design a lookahead-aware scheduler in LLM serving systems to support these asynchronous requests alongside latency-critical workloads with controlled interference. We implement our approach to support representative context engineering strategies and integrate it into existing agent frameworks and LLM serving systems. Experiments show that our approach effectively eliminates transformation overhead and reduces TTFT by up to 24.5×.

1 Introduction

Large language model (LLM)-based agents [3, 21, 31, 48, 57, 60] are increasingly deployed in real-world applications, where they execute multi-turn workflows with interleaved model inference, tool usage, and user interaction, as shown in Figure 1(a). In such scenarios, the context continuously accumulates conversation history, tool outputs, and intermediate states, quickly becoming a central factor for both model quality and system efficiency. As Figure 1(b) shows, prior work has identified *context rot* [20], where model reliability degrades as the context grows, even within the supported context window. Meanwhile, longer contexts also incur higher computational cost and GPU memory usage, making effective context management essential for production LLM agents.

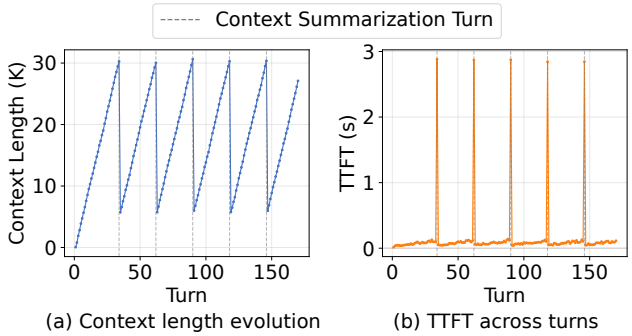


(a) Multi-turn LLM agents (b) LLM performance on repeated words tasks

Figure 1: Without context engineering, the contexts of LLM agents will accumulate rapidly through rounds, leading to the context rot phenomenon. The experimental results in (b) are reported by Chroma [20].

To mitigate context rot, production agents [23,29,34,37,49] adopt a variety of context engineering strategies to control the context length. Common context engineering techniques include reduction [28], which truncates or summarizes historical information; offloading [23], which moves parts of the context such as tool outputs to file systems; and isolation [13,28], which launches sub-agents with separate and clean contexts. As the number of interaction turns increases, the accumulated context grows continuously, while these strategies periodically reduce or reorganize the context to maintain a manageable size. As shown in Figure 2(a), triggering operations such as offloading or summarization effectively controls context growth over time. Context engineering has become a fundamental component of agent harness development and is critical to the effectiveness of agent frameworks.

However, although context engineering improves agent



(a) Context length evolution (b) TTFT across turns

Figure 2: By periodically applying context summarization, LLM agents effectively control the context length, but introduce significant context transformation overhead.

accuracy and robustness, it introduces a new performance challenge. From a system perspective, these techniques fundamentally perform context transformation, where the original context is mapped to a new representation. This transformation invalidates the existing KV cache, forcing the system to recompute it through a new prefill stage. As a result, each transformation incurs additional latency, which we refer to as *context transformation overhead*. As illustrated in Figure 2(b), when such transformations are triggered, the time-to-first-token (TTFT) increases significantly, particularly affecting tail latency. In latency-sensitive applications such as real-time simulation [46, 68], this increase in tail TTFT can lead to unacceptable violations of service-level objectives (SLOs), severely degrading user experience.

To address the context transformation overhead, we first analyze commonly used context engineering techniques and make a key observation: most context transformations exhibit a *segment-decomposable* property. Intuitively, this property implies that the transformation of a context can be decomposed into transformations over its individual segments, such that applying the transformation to each segment independently and then concatenating the results yields the same outcome as transforming the entire context at once. This means that the transformation of each segment is independent of the tokens that follow it, enabling opportunities to perform transformations in advance. We formalize this segment-decomposable property in Section 3 and show that it holds for various context engineering strategies.

Based on this insight, we propose a *lookahead programming model* that eliminates context transformation overhead by decoupling transformation from the critical path. The key idea is to maintain a lookahead context alongside the working context and asynchronously compute the KV cache of transformed contexts in advance. When a context transformation is required, the system can directly replace the KV cache without performing a new prefill. For example, in some agents that offload all tool outputs to file systems once a context limit is reached, the transformed context after each tool call is independent of future tokens. This allows the system to precompute the corresponding KV cache in the lookahead stream after each tool call and seamlessly switch to it when reaching context limit. We demonstrate how various context engineering techniques can be expressed within this programming model and benefit from reduced overhead in Section 3.

While lookahead computation reduces transformation latency, it introduces potential interference with foreground requests. Lookahead requests may compete with the main context generation or with requests from other users in a shared serving environment. To address this challenge, we design a *lookahead-aware request scheduler* that minimize interference under both prefill-decode aggregation and disaggregation settings. Specifically, we split lookahead requests into dynamic chunks and propose SLO-aware lookahead and latency-critical request co-batching strategies. These ensure that lookahead computation is performed only when it does not compromise the latency guarantees of other requests.

We implement our approach by integrating optimized lookahead-based context engineering strategies into popular agent frameworks, including MiniAgent [38], OpenClaw [49], LangChain [29], LlamaIndex [34], and AutoGen [37]. Our evaluation demonstrates that the proposed methods significantly improve tail TTFT by effectively hiding context transformation overhead, leading to a more smooth TTFT profile.

In summary, this paper makes the following contributions:

- We identify and characterize the context transformation overhead introduced by context engineering in LLM agent workloads.
- We formalize the segment-decomposability property of commonly used context transformations.
- We propose a lookahead programming model that generalizes across context engineering strategies and eliminates transformation overhead, along with SLO-aware scheduling techniques to mitigate request interference.
- We conduct comprehensive experiments across strategies and agent frameworks, demonstrating up to 24.5× improvement in transform-point TTFT.

2 Background and Motivation

LLM agents and context engineering. LLM-based agents [3, 17, 57, 60] have emerged as a dominant paradigm for solving complex, multi-step tasks, and are widely adopted in both research and production systems [4, 6, 13, 21, 45, 53, 59]. Instead of a single forward pass, agents iteratively interact with the environment through interleaved LLM calls, tool invocations, and human feedback. Each step produces new observations that are appended to the context, forming a growing execution trace over time.

As a result, agent workloads naturally consist of long, multi-turn sessions with rapidly expanding context. This growth introduces two key challenges. First, the context may exceed the model's context window, forcing the system to discard, compress, or externalize information. Second, even within the limit, models often suffer from *context rot* [20], where performance degrades as the context becomes longer and noisier, as illustrated in Figure 1(b).

These challenges make context management a first-class concern in agent systems. Recent studies and production deployments show that agent performance depends critically on how the context is constructed, maintained, and transformed. This has led to the emergence of *context engineering* [5,23,28] as a fundamental component of modern agent frameworks.

To manage context growth, agent frameworks apply a range of context engineering strategies [5, 9, 11, 17, 23, 28]. For example, some systems offload past interactions or tool outputs to external storage, others summarize long histories to reduce context length, and many isolate sub-tasks into separate sub-agent contexts with dedicated prompts. These transformations help control context size and improve robustness, but they also introduce execution overhead. Each transformation modifies the input prefix and requires recomputing the KV cache, which lies on the critical path of inference and can lead to noticeable TTFT spikes, as shown in Figure 2.

Agent serving systems. The serving stack for LLM agents typically consists of two loosely coupled layers. At the frontend, agent frameworks [4, 29, 34, 38, 49] define the agent harness, including control flow, tool usage, and context management policies. These frameworks focus on expressiveness and task-solving capability, enabling developers to compose complex agent behaviors.

At the backend, LLM serving systems such as vLLM [27] and SGLang [64] optimize model execution efficiency. They employ techniques such as batching and scheduling [2, 43, 62, 65, 66], kernel optimization [10, 12, 25, 40, 61], and KV cache management [1, 27, 44, 58] to improve throughput and reduce latency across concurrent requests.

In this work, we improve the execution efficiency of agent systems under context engineering. At the agent framework level, we provide a lookahead programming model that enables context transformations to be expressed in a form that can be executed asynchronously without changing their logic. At the LLM serving system level, we optimize the scheduler to support these lookahead requests, allowing them to run alongside regular requests with controlled interference.

3 Lookahead Programming Model for Context Transformation

3.1 Segment-Decomposable Transformation

In this section, we formalize a common structural property that enables more efficient context engineering execution. In particular, we view context engineering as a transformation process over the accumulated context. Let C denote the current context of the agent, and let $T(\cdot)$ denote a transformation function that is triggered when certain conditions are met, such as the context length exceeding a predefined threshold. The transformed context $T(C)$ is then used for subsequent LLM invocations to maintain generation quality.

In existing systems, such transformations are typically executed synchronously on the critical path of generation, introducing non-trivial overhead. First, applying T invalidates the KV cache of the original context, requiring recomputation for the transformed input. Second, the transformation itself incurs additional cost, which may involve file system operations, external storage access, or auxiliary model calls such as summarization. As a result, synchronous transformation increases latency, particularly affecting TTFT in production systems. This motivates the need to execute context transformations ahead of time, without blocking ongoing generation.

To understand when such asynchronous execution is possible, consider the example of context offloading. As illustrated in Figure 4, when the context exceeds a threshold, agent frameworks may offload certain components, such as tool outputs, to file systems, thereby reducing the working context length. In this process, the transformation applied to each tool output is independent of others. Once a tool output is produced, its transformed representation is fully determined and does not depend on future tokens or subsequent interactions. Therefore, its transformation can be computed immediately, rather than waiting for the global offloading trigger.

This observation motivates the notion of *segment-decomposability*. Let the context C be partitioned into a sequence of segments $\{S_1, S_2, \ldots, S_n\}$. A transformation function T is segment-decomposable if there exists such a partition where each segment can be transformed independently, and the overall transformation can be expressed as

$$T(C) = T(S_1) \parallel T(S_2) \parallel \cdots \parallel T(S_n),$$

where $\parallel$ denotes concatenation. Under this property, each segment can be transformed as soon as it becomes available, enabling asynchronous execution in advance and avoiding increases in TTFT.

In Sections 3.3-3.5, we show that widely used context engineering strategies, including offloading, reduction, and isolation, naturally exhibit this property, which enables their transformation overhead to be hidden via lookahead execution.

3.2 Lookahead Programming Model

Building on segment-decomposability, we design a *lookahead programming model* that expresses context engineering strategies in a form that exposes transformation opportunities to the runtime. The key idea is to maintain a *lookahead context* alongside the main working context, as illustrated in Figure 3. The model consists of a strategy interface together with a

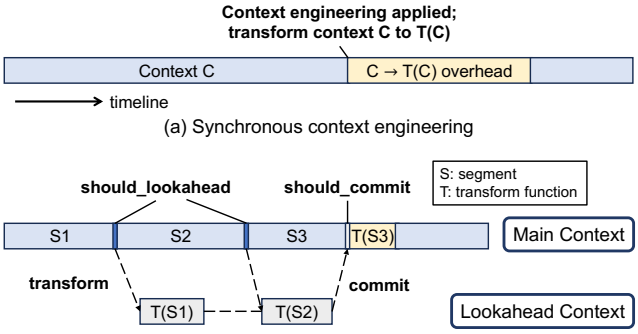


Figure 3: Execution timeline of the lookahead programming model. The main stream proceeds normally while the lookahead stream asynchronously computes the transformed segments' KV cache in advance. At commit time, the precomputed result is spliced.

runtime that asynchronously executes lookahead transforms and finally commits their results into the main context. We present their design in the rest of this section.

State abstractions. The model maintains two state objects corresponding to the working context and the lookahead context, as shown in Listing 1. `MainState` captures the current working context, including the full message list and per-strategy metadata such as token counts and limits, which are used to determine when to trigger different actions. `LookaheadState` tracks the progress of lookahead execution. It contains (1) `transformed`, the accumulated transformed segments constructed incrementally across processed segments; (2) `last_segment_end`, which marks the end of the prefix that has already been incorporated into the lookahead state; and (3) `strategy_data`, a per-strategy mutable state used to store auxiliary information such as file handles for offloading or intermediate summaries.

Strategy interface. As shown in Listing 1, each context engineering strategy implements the following methods:

- `should_lookahead() → bool`: invoked at segment boundaries to determine whether the newly available segment should be transformed asynchronously. Different strategies instantiate this condition differently.

- `transform()`: incrementally updates the transformed prefix using the current segment. The function mutates `la_state` to reflect updated progress. By segment-decomposability, the transformation depends only on the current segment and the existing lookahead state, and is independent of any future tokens.

- `should_commit() → bool`: determines whether the precomputed transformed prefix should be committed into the working context. This condition is typically triggered when the context exceeds the token limit.

- `should_promote() → bool`: provides an early signal to increase the scheduling priority of pending lookahead requests. By default, this condition aligns with `should_commit`, but strategies may override it to promote lookahead execution before the commit point, reducing the likelihood of misses.

Listing 1: Strategy interface of the programming model

```python
@dataclass
class MainState:
    messages: list[Message]
    strategy_data: Dict

@dataclass
class LookaheadState:
    transformed: List[Message]
    last_segment_end: int
    strategy_data: Dict

class LookaheadStrategy(ABC):
    def __init__(self):
        self.main = MainState(...)
        self.la = LookaheadState(...)

    @abstractmethod
    def should_lookahead(self) -> bool: ...

    @abstractmethod
    def transform(self): ...

    @abstractmethod
    def should_commit(self) -> bool: ...

    def should_promote(self) -> bool:
        return self.should_commit()
```

Execution runtime. The runtime maintains two concurrent streams, as illustrated in Figure 3. The *main stream* executes the agent loop as usual, while the *lookahead stream* processes segments asynchronously.

At each segment boundary, the runtime invokes `should_lookahead`. If triggered, it extracts the newly available segment and submits a `transform` call as a lookahead request. Although `transform` is defined over a single segment, its execution introduces two challenges. First, LLM calls within `transform` require KV cache construction that depends on the previously transformed prefix. Second, the lookahead state must be updated incrementally to reflect the progressive construction of the transformed context. To preserve this dependency and for simplicity, lookahead requests are enqueued and executed in order, so that each request observes the appropriate prefix state. All LLM requests issued within the `transform` function are marked as *lookahead requests* and scheduled outside the latency-critical path. We discuss their scheduling in Section 4.

When `should_commit` fires, the runtime attempts to commit the lookahead result by splicing the transformed prefix into the working context. If the corresponding lookahead computation has completed, the runtime directly replaces

the prefix in the working context with the transformed result, thereby avoiding blocking transformation on the critical path. Otherwise, the runtime promotes the priority of pending lookahead requests so that the scheduler can expedite their execution. Such promotion can also happen proactively before the commit point if `should_promote` is satisfied.

Listing 2 shows how the programming model integrates into a standard agent loop. The runtime encapsulates the complexity of lookahead scheduling and result management, while developers only need to implement the strategy interface and insert lightweight hooks into the agent loop.

Listing 2: Agent loop with lookahead context engineering

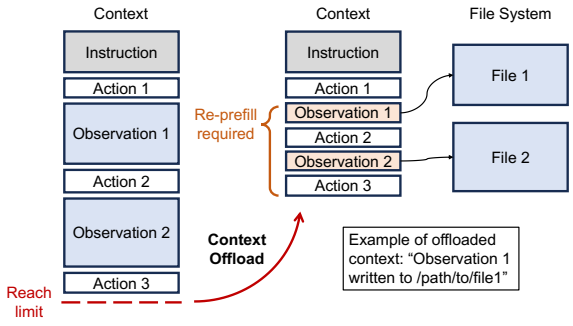
```
strategy = MyLookaheadStrategy(...)
runtime  = LookaheadRuntime(strategy)

messages = [system_prompt, user_query]
while not task_done:
    # Commit hook: if context budget is exceeded, await
    # lookahead transforms and promote if necessary.
    # Then splice the transformed prefix:
    #   main.messages <= la.transformed
    #     ++ T(messages[la.last_segment_end:])
    messages = runtime.on_commit(messages)

    response = llm.generate(messages)
    messages.append(response)
    if response.has_tool_call:
        obs = execute_tool(response.tool_call)
        messages.append(obs)

    # Segment hook: extract new segment since last
    # trigger and enqueue a background transform.
    # Completed result is appended to la.transformed.
    runtime.on_segment_boundary(messages)
```

In the following sections, we instantiate this programming model with commonly used context engineering strategies, showing how each strategy satisfies segment-decomposability and maps naturally onto the lookahead execution model.

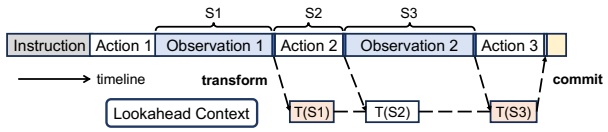
3.3 Offloading

Context offloading [9, 23] rewrites the context by moving observations (e.g., tool execution results, environment states) to external storage and replacing them with lightweight references. As illustrated in Figure 4(a), when the context approaches a predefined limit, earlier observations are written to the file system. Their in-context representations are replaced with pointers such as "Observation 1 written to /path/to/file1", which triggers a new blocking prefill request. This design preserves the full information in a restorable form while reducing the active context length. Unlike lossy techniques such as summarization, offloading maintains exact fidelity and allows the agent to read file contents when needed.

Offloading fits naturally into our segment-decomposable abstraction. The decision to offload depends only on the current context up to a segment boundary (e.g., after an observation), and each segment can be transformed independently of future segments. This property enables the system to incrementally construct transformed prefixes ahead of time and



(a) Illustrative example of context offloading



(b) Lookahead-optimized context offloading

Figure 4: Offloading transforms bulky observations (e.g., tool outputs) into compact references to external storage. Since each completed observation can be rewritten independently, the transformed KV cache can be prepared ahead of time.

overlap transformation with ongoing execution.

As shown in Figure 4(b), `should_trigger` is evaluated after each segment is appended to determine whether the current segment should be offloaded. Once triggered, `transform` incrementally rewrites the prefix by offloading completed segments to external storage and replacing them with references in the lookahead stream, while updating the lookahead state to track the materialized outputs. In parallel, `should_commit` is evaluated as the main execution progresses; when the context reaches the offloading point, the system replaces the corresponding prefix in the main context with the transformed prefix produced by the lookahead execution.

By preparing the transformed prefix in advance, the system avoids performing the transformation and re-prefill synchronously on the critical path, effectively hiding the overhead of context offloading.

3.4 Reduction

Context reduction [9, 28, 30] controls context growth by discarding or compressing less relevant history while retaining information that is most useful for future reasoning. Two representative forms are truncation and summarization.

Truncation. Truncation strategies directly discard historical contents to control context growth. The most common and simple strategy is keep-recent-K [11, 17], which retains only the most recent K turns while discarding older ones. As shown in Figure 5(a), when a new turn is appended and the number of turns exceeds the limit, the oldest turns are removed. Although the contents in Turns 3 and 4 do not change,

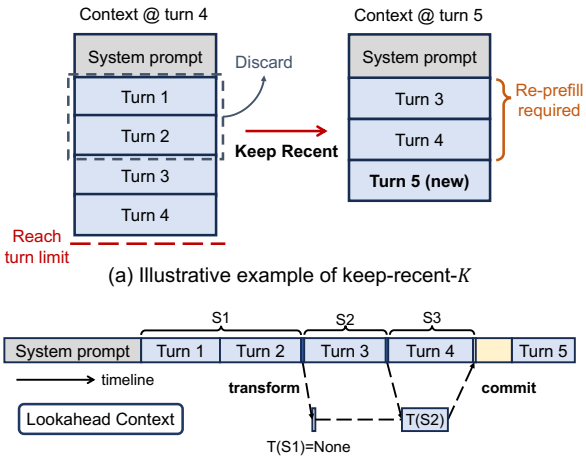


Figure 5: Keep-recent-K reduction keeps only the most recent K context segments. The transform is local to the retention boundary, which meets the segment-decomposable property.

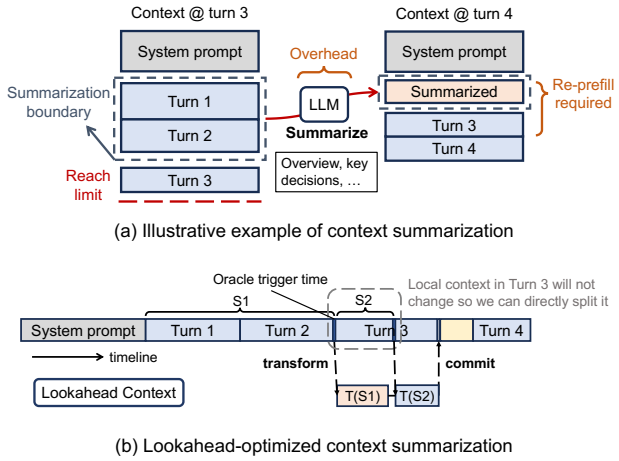


Figure 6: Summarization compresses the prefix into a compact synopsis. The summary depends only on the completed history, which makes it amenable to lookahead execution.

a new prefill is still required to rebuild the KV cache for subsequent generations. This approach is simple yet effective, as recent interactions are typically the most relevant for next-step reasoning in agent workloads [35].

The keep-recent-K example also fits into the segment-decomposable abstraction. Each turn can be treated as a segment, and whether a segment is retained depends only on its relative position within the window. Since removal decisions do not depend on future segments, the transformed prefix can be constructed incrementally.

As illustrated in Figure 5(b), `should_trigger` is evaluated after each new turn to determine whether the window size exceeds the limit. Once triggered, `transform` incrementally drops outdated segments and constructs the truncated prefix, computing the corresponding KV cache in advance. When the window boundary shifts, the system detects the commit signal and replaces the corresponding prefix with the truncated version produced by lookahead execution.

Summarization. Another form of reduction replaces a span of past context with a compact summary generated by an LLM. As shown in Figure 6(a), when the context reaches a predefined limit, a subset of earlier turns is summarized into a shorter representation (e.g., key decisions or high-level descriptions), which is then inserted back into the context. This transformation reduces context length while preserving approximate semantic information.

Summarization also exhibits segment-decomposability, but with a coarser granularity. In practice, agents typically preserve recent turns verbatim and only summarize earlier history to maintain fidelity. As a result, summarization only applies to the completed segments before the summarization boundary, making it independent of future context.

Determining the exact summarization boundary may require observing the full context (e.g., when summarizing all but the most recent N turns). To enable lookahead execution, we use a *soft trigger threshold* that is slightly smaller than the actual summarization limit. When this soft threshold is reached, `should_trigger` initiates a lookahead summarization request based on the current prefix. Although the resulting boundary may not exactly match the final one, the discrepancy is bounded by the gap between the soft and hard thresholds. In practice, this introduces negligible impact when the total context is long, and may even improve fidelity by summarizing slightly less content.

3.5 Isolation

Context isolation [5, 28] separates a task into a new execution context, typically by launching a sub-agent with its own system prompt and tool set. This design helps reduce interference from irrelevant history, improves modularity, and allows the sub-agent to operate with a specialized prompt and tool configuration tailored to the delegated task. As shown in Figure 7(a), when the main agent delegates a task, it first generates an instruction describing the sub-task, and then switches to a clean context for the sub-agent. This new context consists of the sub-agent's system prompt and the generated instruction, and requires a fresh prefill before execution. This process can be viewed as a context transformation that replaces the main-agent prefix with a new sub-agent context.

Isolation can also be expressed within the segment-decomposable abstraction. We treat the point where the main agent decides to delegate a sub-agent as the first segment boundary, and transform it into the sub-agent system prompt in advance. After that, the main agent generates instructions for the sub-agent. This instruction generation, however, is not

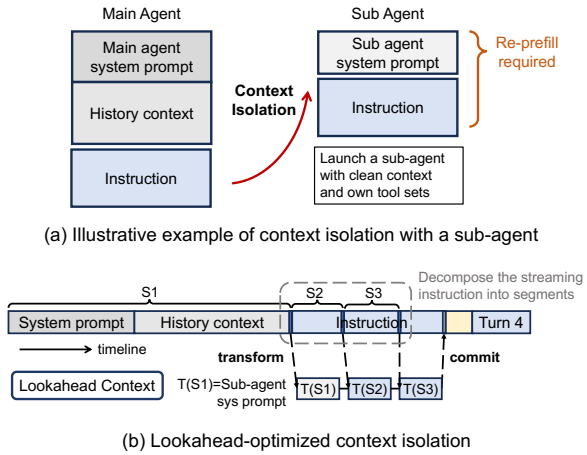


Figure 7: Context isolation launches a sub-agent with a clean context. By decomposing instruction generation into segments, the sub-agent context can be prepared ahead of time.

naturally decomposed for context transformation. Treating it as a monolithic output delays sub-agent context construction until decoding completes, leading to performance for long instructions.

To address this issue, we reformulate instruction generation as a *streaming* process and partition it into fine-grained segments. Each segment corresponds to a chunk of newly generated tokens and depends only on the previously generated prefix, making it compatible with segment-decomposability. This enables the system to incrementally construct the sub-agent context alongside decoding, rather than waiting for the full instruction. Since instruction generation is a relatively long-running auto-regressive decoding process, it creates sufficient opportunity to prepare the sub-agent KV cache incrementally. As a result, most of the prefill cost can be amortized and hidden before the context switch occurs. FlashAgents [14] also leverages similar streaming ideas to accelerate multi-agent workflows.

As illustrated in Figure 7(b), `should_trigger` is activated when the system decides to launch a sub-agent. During instruction generation, the decoding stream is partitioned into segments (S2, S3), each corresponding to a chunk of the instruction. For each segment, `transform` incrementally constructs the target sub-agent context by appending the generated instruction chunk to the sub-agent system prompt and preparing the corresponding KV cache. Since instruction generation is typically slower than prefilling, most of the sub-agent context can be prepared in advance and hidden from the decoding process.

Finally, `should_commit` is evaluated when the instruction generation completes. At this point, the system replaces the main-agent context with the constructed sub-agent context and performs a final short prefill to complete the transition. By preparing the majority of the sub-agent context in advance, the system significantly reduces the blocking prefill overhead of context isolation.

4 Lookahead-Aware Request Scheduler

Co-serving lookahead requests with other requests introduces interference that can degrade serving latency. On prefill-only instances [43, 44, 66], lookahead requests consume service time and increase the queuing delay of other requests. In prefill-decoding (PD) co-located deployments [2, 19, 25], including lookahead chunks in a batch increases its execution time, delaying subsequent iterations. As a result, both time-to-first-token (TTFT) and time-between-tokens (TBT) can be negatively affected.

These effects are particularly problematic for requests on the critical path of agent execution, including prefills and decode iterations in the main context. Their latency directly determines TTFT and TBT, and we refer to them as *latency-critical* (LC) requests.

To mitigate such interference, SmoothAgent treats lookahead requests as *best-effort* (BE) jobs. Lookahead requests have inherently looser timing requirements: they only need to complete before their corresponding commit point to fully realize their benefit, and can otherwise fall back to synchronous execution without incurring additional overhead compared to the baseline. Therefore, BE jobs are executed only when sufficient slack exists and must never degrade the latency guarantees of LC requests. Concretely, LC requests always take priority, and lookahead computation is admitted only if it does not violate TTFT or TBT constraints.

Enforcing this policy requires accurately estimating the latency impact of co-batching lookahead and LC requests. We first present a performance model for batch execution latency, and then describe SLO-aware co-batching strategies for both PD disaggregated and co-located deployments.

Performance model. Some prior systems [2, 18] use token budgets based on SLO requirements to enable stall-free piggyback execution, modeling batch latency solely as a function of the total number of tokens. However, this approximation is insufficient in agentic workloads, where requests often carry long KV caches. In this regime, attention cost becomes a dominant factor and depends not only on the number of query tokens but also on the KV cache length. Therefore, the scheduler must explicitly model both effects to make reliable admission decisions.

We model the latency of a mixed batch B by decomposing transformer execution into components with distinct scaling behaviors. The key observation is that FFN layers scale with the number of tokens, while attention layers scale with both token count and KV-cache length.

GEMM. All tokens in the batch, including decode tokens and prefill chunk tokens, are processed by the same dense operators in each transformer layer, including the projection matrices in attention (QKV and output projections) and the

FFN layers. Let

$$M = |B_{\text{decode}}| + \sum_{j \in B_{\text{prefill}}} q_j,$$

denote the total number of forward tokens in the batch. The GEMM cost $T_{\text{GEMM}}(M)$ is non-linear in M: for small M, execution is memory-bound and exhibits near-constant latency, while for larger M, it becomes compute-bound and scales approximately linearly. We therefore model T_{GEMM} using an offline-profiled lookup table with interpolation.

Decode attention. Each decode request contributes a single query token that attends to its full KV cache of length L_j. This operation is memory-bandwidth bound and its cost scales linearly with the total KV tokens accessed:

$$\alpha_{\text{d}} \sum_{j \in B_{\text{decode}}} L_j,$$

where α_{d} is a profiled coefficient.

Prefill attention. Each prefill chunk j contains q_j query tokens attending to a KV cache with prefix length prefix_j. In contrast to decode attention, prefill attention is compute-bound due to the larger number of query tokens per request, and we model its cost based on total attention work. Due to the causal mask, the i-th query token attends to $\text{prefix}_j + i + 1$ tokens, leading to

$$A_j = \sum_{i=0}^{q_j-1} (\text{prefix}_j + 1 + i) = q_j \cdot \text{prefix}_j + \frac{q_j(q_j+1)}{2}.$$

The total prefill attention cost is therefore

$$\alpha_{\text{p}} \sum_{j \in B_{\text{prefill}}} A_j,$$

where α_{p} is a profiled coefficient.

Combining the three components, the estimated batch latency is

$$\begin{aligned} \text{EstBatchLatency}(B) = \quad & T_{\text{GEMM}}(M) \\ & + \alpha_{\text{d}} \textstyle\sum_{j \in B_{\text{decode}}} L_j \\ & + \alpha_{\text{p}} \textstyle\sum_{j \in B_{\text{prefill}}} A_j. \end{aligned} \tag{1}$$

This model enables accurate estimation of the latency impact of co-batching decode and prefill requests, and is used to guide SLO-aware scheduling decisions.

Slack-aware batching for PD disaggregation. In PD disaggregated systems [43, 44, 66], prefill and decode are served by separate instances. Lookahead requests correspond to prefill operations and are handled by prefill instances.

Our goal is to piggyback lookahead requests on prefill instances without violating the TTFT constraints of other requests. To enable fine-grained scheduling, we divide prefill requests into small chunks [2, 19, 44] and treat each chunk as a schedulable unit. The key idea is to treat lookahead execution as slack-driven computation. Instead of explicitly reserving resources for BE requests, we derive a time budget from the TTFT constraints of LC requests and admit BE chunks only within this budget. This reduces the scheduling problem to a simple question: how much additional latency can be introduced without violating any LC deadline.

Algorithm 1 Prefill Batch Scheduling (PD Disaggregated)

Input: LC queue Q_{LC}, BE queue Q_{BE}, LC schedule algorithm $\mathcal{A}_{\text{LC}}$, current time t_{now}
Output: batch to be executed $B = B_{\text{LC}} \cup B_{\text{BE}}$

1: // Schedule LC reqs based on the original algorithm
2: $Q_{\text{LC}} \leftarrow \text{Schedule}(Q_{\text{LC}}, \mathcal{A}_{\text{LC}})$
3: // Get minimum slack across LC queue
4: $p \leftarrow 0$; $s_{\min} \leftarrow +\infty$
5: **for** each request $r \in Q_{\text{LC}}$ **do**
6: $p \leftarrow p + \text{EstPrefillLatency}(r)$
7: $s_j \leftarrow \left(r.t_{arrival} + r.SLO_{TTFT}\right) - t_{now} - p$
8: $s_{\min} \leftarrow \min(s_{\min}, s_j)$
9: **end for**
10: $t_{budget} \leftarrow \max(0, s_{\min})$
11: // Construct batch
12: $B_{\text{LC}} \leftarrow \text{NextChunks}(Q_{\text{LC}})$
13: $B_{\text{BE}} \leftarrow \emptyset$
14: **for** each candidate chunk $c \in Q_{\text{BE}}$ **do**
15: **if** $\text{EstBatchLatency}(B_{\text{LC}} \cup B_{\text{BE}} \cup \{c\}) > t_{budget}$ **then**
16: **break**
17: insert c into B_{BE}
18: **end for**
19: **return** $B_{\text{LC}} \cup B_{\text{BE}}$

As shown in Algorithm 1, the scheduler first applies the original LC scheduling policy (line 2), such as first come first serve (FCFS), and then computes the available slack from LC requests. It iterates over the LC queue (line 5), accumulating the expected prefill latency of preceding requests, and derives a slack value for each request based on its TTFT deadline. The minimum slack determines the time budget t_{budget} (line 10).

Using this budget, the scheduler constructs a batch by first selecting LC chunks (line 12), and then greedily admitting BE chunks (line 14). Each candidate chunk is included only if the resulting batch latency does not exceed t_{budget}, ensuring that BE execution does not delay any LC request.

TBT-constrained batching for PD co-location. In PD co-located systems [2, 19, 25], prefill and decode share the same instance and are executed within a hybrid batch. In this setting, decode requests are subject to TBT SLOs, since each iteration directly affects token generation latency.

Unlike the PD disaggregated setting, where slack is derived from queued requests, TBT constraints in co-located systems are enforced at the granularity of each iteration. Each batch is constructed to satisfy the latency bound δ, and additional work

Algorithm 2 Hybrid Batch Scheduling (PD Co-Located)

Input: decode queue Q_{dec}, LC prefill queue Q_{LC}, BE prefill queue Q_{BE}, TBT SLO δ
Output: batch to be executed B

1: $B \leftarrow \emptyset$
2: // Schedule decode requests
3: **for** each request $r \in Q_{\text{dec}}$ **do**
4: **if** EstBatchLatency$(B \cup \{r\}) > \delta$ **then break**
5: insert r into B
6: **end for**
7: // Schedule LC prefill requests
8: **for** each chunk candidate $c_{LC} \in Q_{\text{LC}}$ **do**
9: **if** EstBatchLatency$(B \cup \{c_{LC}\}) > \delta$ **then break**
10: insert c_{LC} into B
11: **end for**
12: // Schedule BE prefill requests
13: **for** each chunk candidate $c_{BE} \in Q_{\text{BE}}$ **do**
14: **if** EstBatchLatency$(B \cup \{c_{BE}\}) > \delta$ **then break**
15: insert c_{BE} into B
16: **end for**
17: **return** B

is admitted only if it preserves this constraint. This enables lookahead execution to be interleaved with decode without affecting token generation latency.

To preserve TBT guarantees, as shown in Algorithm 2, the scheduler constructs the batch in a priority order. It first admits decode requests (line 3), ensuring that the core token generation loop is not delayed. It then schedules LC prefill chunks (line 8), followed by BE lookahead chunks (line 13). At each step, a candidate is admitted only if the resulting batch latency remains within the TBT bound δ.

This ordering, combined with the latency constraint, ensures that decode and LC requests are never delayed by BE execution, while allowing the scheduler to exploit slack within each iteration.

Discussion on API-based serving. The best-effort abstraction for lookahead requests also enables flexible API design. Service providers can expose lookahead requests as a lower-priority class with discounted pricing, since they consume only spare capacity and carry no latency guarantees. Furthermore, after a lookahead context commits, subsequent generation can directly reuse the cached prefix tokens prepared by lookahead execution, which can be billed at a reduced rate. This creates a natural incentive for users to adopt lookahead context engineering strategies, aligning system efficiency improvements with reduced serving cost.

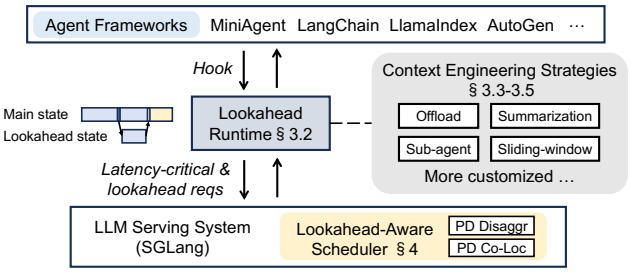


Figure 8: System overview of SmoothAgent.

5 Implementation

We implement SmoothAgent, a system that enables lookahead execution for context transformation in LLM-based agents. As illustrated in Figure 8, the SmoothAgent runtime sits between agent frameworks (e.g., MiniAgent [38] and LangChain [29]) and the underlying LLM serving system to orchestrate lookahead execution.

SmoothAgent exposes a unified interface for supporting various context engineering strategies, such as offloading, summarization, and sub-agent isolation. Agent frameworks interact with SmoothAgent through a simple hook mechanism, allowing lookahead requests to be issued alongside latency-critical requests without modifying application logic. Internally, the runtime of SmoothAgent maintains both the main execution state and a separate lookahead state, and manages their interaction through the lookahead programming model introduced in Section 3.2.

To support lookahead requests, SmoothAgent integrates with the LLM serving system, SGLang [64] v0.5.9, and incorporates a lookahead-aware scheduler. The scheduler supports both prefill-decode co-located and disaggregated deployment settings, enabling best-effort execution of lookahead workloads while preserving latency SLO guarantees for foreground requests. This design allows SmoothAgent to overlap context transformation with normal inference, effectively hiding transformation overhead from the critical path.

6 Evaluation

6.1 Experimental Setup

Hardware. We evaluate SmoothAgent on NVIDIA H100 GPUs with 80GB HBM, connected via NVLink within a node. Host machines are equipped with AMD EPYC CPUs.

Models. We evaluate two representative open-source LLMs: Qwen3-8B and Qwen3-32B [56]. Both models use grouped-query attention (GQA) [52] and dense feed-forward layers, with a maximum context length of 32K tokens.

Deployment configurations. We evaluate two deployment settings. In the *PD co-located* setting, prefill and decode share the same GPU instance. We run Qwen3-8B on a single H100

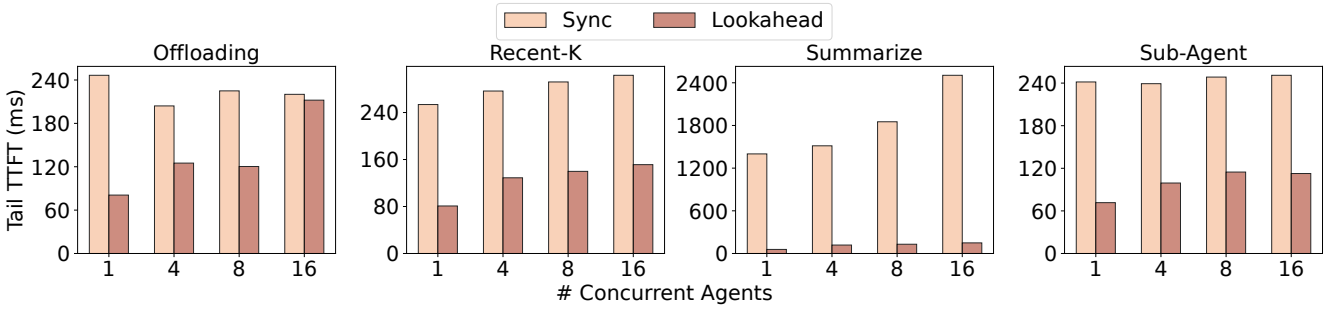


Figure 9: Transform-point TTFT for each strategy on Qwen3-8B (PD co-located) at concurrency levels 1, 4, 8, and 16. SmoothAgent eliminates the transformation-induced TTFT spike across all strategies and concurrency levels.

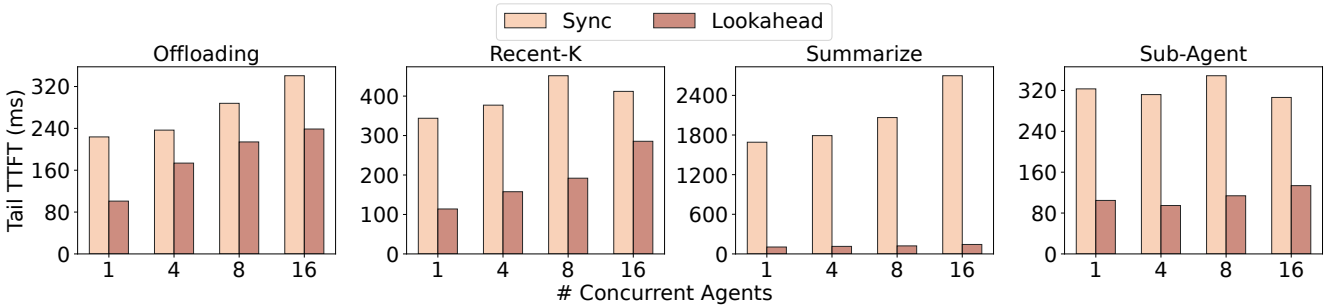


Figure 10: Transform-point TTFT for each strategy on Qwen3-32B (PD co-located, TP=4 across four H100 GPUs).

GPU and Qwen3-32B on four H100 GPUs with tensor parallelism (TP=4), varying the number of concurrent agents from 1 to 16. Rather than modeling request arrivals, we adopt a fixed-concurrency configuration: each agent runs a long-lived session with causally interleaved prefill and decode requests, which the arrival-rate-based models for stateless workloads do not capture. Agents are launched at staggered start times to avoid burst arrivals, and each agent runs in an isolated process to eliminate event-loop interference. Due to large KV cache footprints at long contexts, we do not scale concurrency further within a single instance.

In the *PD disaggregated* setting, prefill and decode run on separate instances. Prefill instances serve prefill requests and maintain an SGLang radix cache [64]; decode instances perform token generation. KV caches are transferred between instances via NIXL [39].

Workloads. Each agent executes a multi-step code analysis task. At each step, the agent reads source code from the MiniAgent [38] codebase via shell commands (e.g., `head`, `tail`, `sed`) and produces an analysis response, generating realistic workloads with interleaved LLM calls and tool executions and causing the context to grow monotonically. Each run consists of 28 steps, with approximately 500–600 tokens added per step. All agents receive unique prompts to prevent radix cache sharing from suppressing context growth.

Baselines. We compare against synchronous context engineering strategies, denoted as *sync*, which are widely adopted in existing agent frameworks. In these systems, context transformations are triggered reactively and executed on the critical path, blocking subsequent LLM generation. To ensure fair comparison, the baselines implement the same context engineering strategies as SmoothAgent, but without lookahead execution. Both the baselines and SmoothAgent use SGLang [64] as the LLM serving backend.

6.2 Lookahead Strategy Evaluation

We implement four context engineering strategies based on MiniAgent [38], a lightweight agent framework that enables controlled evaluation without framework-specific optimizations. The strategies are offloading, keep-recent-K, summarization, and sub-agent isolation, following the strategy descriptions in Section 3. Each strategy is evaluated independently. We focus on tail TTFT, especially at transformation points where synchronous execution introduces blocking overhead. This allows us to directly evaluate whether lookahead execution removes the latency spikes associated with context transformation.

PD co-located. Figures 9 and 10 show results on Qwen3-8B and Qwen3-32B in the PD co-located setting. SmoothAgent consistently reduces tail TTFT across all four strategies and both model scales, with an average reduction of 62.1% and 61.7% on Qwen3-8B and Qwen3-32B, respectively.

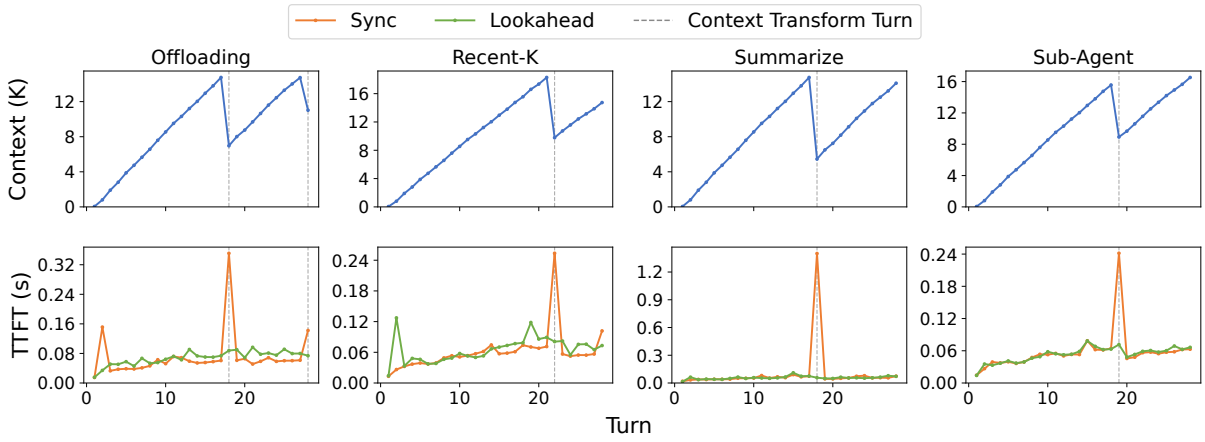


Figure 11: Context length (top) and per-turn TTFT (bottom) for a single Qwen3-8B agent under each strategy. Synchronous transformation produces a TTFT spike at each trigger; SmoothAgent absorbs the transformation cost across preceding turns and delivers an uninterrupted TTFT profile.

Among these strategies, summarization yields the largest gain, with up to 24.5× TTFT improvement. Unlike other strategies, summarization incurs both an additional LLM generation to produce the summary and a full re-prefill of the resulting context, leading to substantially higher overhead. SmoothAgent moves both operations into the lookahead stream, effectively hiding this compound cost.

In contrast, offloading shows smaller improvements, especially under high concurrency. This is because once earlier segments have been offloaded, subsequent transformations operate on already compact representations and incur little additional transformation cost. As a result, the overhead of synchronous execution diminishes over time, reducing the relative benefit of lookahead. In practice, offloading alone does not effectively control context growth after repeated triggers, and is often combined with reduction strategies such as summarization [23].

To understand when transformations are triggered and their impact on latency, we plot the context length and TTFT across turns in Figure 11 for a single agent on Qwen3-8B. For offloading, the transformation is triggered when the context exceeds 15K tokens. The first trigger reduces the context size by approximately half, while subsequent triggers yield diminishing reductions as earlier segments have already been offloaded. For keep-recent-K, the transformation is triggered at turn 22, retaining the most recent 10 turns along with the system prompt. For summarization, the transformation is triggered at 15K tokens, keeping recent turns whose total length exceeds 4K tokens and summarizing the remaining context into 128 tokens. A soft threshold at 11K tokens enables lookahead execution to start summarization ahead of the hard trigger. For sub-agent, the main agent generates approximately 800 tokens as delegation instructions. The sub-agent includes an additional 800-token system prompt and reads around 7K tokens from the codebase at startup.

As shown in the bottom of Figure 11, all strategies introduce noticeable TTFT spikes at transformation points under synchronous execution, despite reducing context growth. By performing transformations ahead of time and amortizing the cost across multiple turns, SmoothAgent eliminates these spikes and produces a smooth TTFT profile.

PD disaggregated. Figure 12 shows results under PD disaggregation with up to 64 concurrent agents. We use four H100 GPUs as prefill instances and four H100 GPUs as decode instances to serve Qwen3-8B. SmoothAgent consistently reduces TTFT at transformation points across all strategies, achieving an average reduction of 56.9%. Compared to the co-located setting, both the baseline and SmoothAgent exhibit higher absolute TTFT due to additional KV cache transfer and routing overhead. Despite this overhead, lookahead execution remains effective and continues to eliminate transformation-induced latency spikes.

Ablation on lookahead-aware scheduler. We compare the lookahead-aware scheduler with SGLang's default scheduler, which does not distinguish lookahead requests from latency-critical requests. At high concurrency, the lookahead-aware scheduler reduces tail TBT by up to 59%. Without explicit prioritization, lookahead requests compete with decode workloads for GPU resources, delaying ongoing decoding batches. Our scheduler mitigates this interference by admitting lookahead work only within available latency slack. We also evaluate the accuracy of the context-aware performance model used in scheduling decisions. Our model achieves 15.4% mean absolute percentage error (MAPE) against measured latencies. The token-budget-only model from prior work [2], which models batch latency solely as a function of token count, achieves 131.6% MAPE because it cannot account for the attention cost growth driven by long context in agentic workloads.

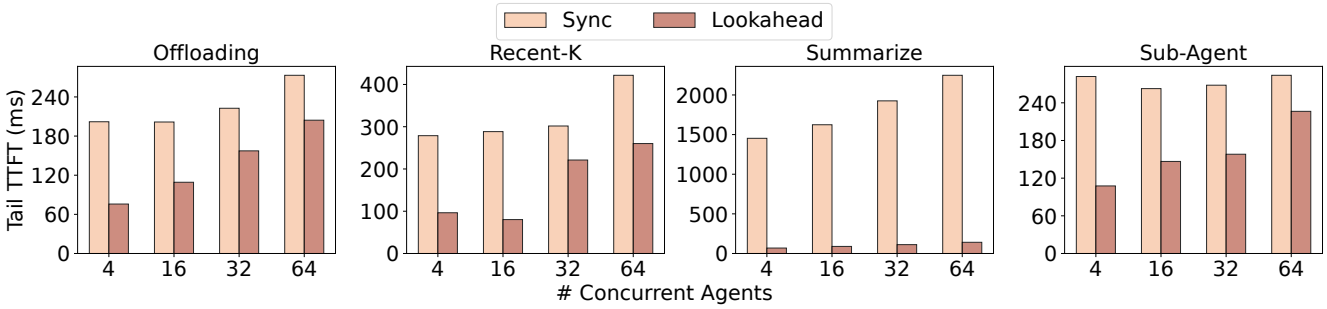


Figure 12: Transform-time TTFT on Qwen3-8B in a PD disaggregated deployment with four prefill and four decode instances.

Agent framework integration. We further integrate SmoothAgent into representative agent frameworks, including LangChain [29] (turn/token-based sliding window and summarization), LlamaIndex [34] (sliding window and summarization), AutoGen [37] (sliding window), and OpenClaw [49] (summarization). These strategies correspond to the context engineering mechanisms natively supported in each framework. In particular, sliding window is another form of context truncation. It maintains a fixed-size (turns or tokens) context by continuously evicting old tokens at each turn, whereas keep-recent-K performs truncation only when the context exceeds a predefined limit, resulting in periodic rather than continuous transformations.

We replace these built-in strategies with their lookahead-optimized counterparts without changing the agent logic, and evaluate them under the same workloads. Across these frameworks, SmoothAgent consistently reduces transformation-point TTFT by 27.5% to 91.5%.

7 Related Work

LLM serving systems. A large body of work has focused on optimizing general-purpose LLM serving systems. vLLM [27] and SGLang [64] are among the most widely used open-source serving engines, and they incorporate a range of important optimizations, including continuous batching [62], paged attention [27], and prefix sharing [64]. Chunked prefill techniques [2, 19] split prefilling into smaller chunks and mix them with decoding, preventing long prefills from blocking decoding TBT while improving GPU utilization in prefill-decode co-located settings. Prefill-decode disaggregation [43, 66] eliminates interference between prefill and decode requests by dispatching them to separate instances, and has become the de facto design in large-scale datacenter LLM serving. More recent work [50, 67] further proposes disaggregating attention and FFN computation in MoE models due to their mismatched computational characteristics, enabling decoupled scaling and more effective use of heterogeneous hardware. Our work optimizes the request scheduler in serving engines to support our lookahead programming model under both PD co-located and disaggregated deployments.

Prefix caching [15, 16, 44, 63, 64] stores KV caches on GPUs or external storage after requests complete, avoiding recomputation when subsequent requests share the same prefix. This capability is particularly important in workloads such as multi-turn conversations. Our lookahead execution also relies on prefix caching to preserve the KV cache of transformed contexts. Another line of work [8, 10, 12, 25, 40, 47, 54, 61] focuses on GPU kernel optimizations for LLM serving. A representative example is FlashInfer [61], which provides efficient kernels for a variety of serving scenarios and has been integrated into many serving frameworks. These kernel-level optimizations are orthogonal to our work.

Agent serving optimization. As agentic workloads become increasingly important, recent research has begun to explore system-level optimizations for agent serving. Parrot [32] and Ayo [51] model static agent workflows as directed acyclic graphs (DAGs) and apply a range of optimizations over the execution graph. In contrast, our work does not assume a static workflow and instead targets more autonomous, multi-turn agents. Other lines of work focus on retrieval-augmented generation (RAG) [22, 24, 33, 58] and on multi-agent simulation [42, 55], both of which differ from our target setting. Autellix [36] mitigates head-of-line blocking by scheduling at the granularity of agent programs rather than individual requests. InferCept [1] and KVFlow [41] improve KV cache management for agentic workloads by adopting application-aware eviction and fetching policies instead of conventional LRU. ThunderAgent [26] and CONCUR [7] identify memory thrashing in multi-turn agents and propose admission control mechanisms based on current system load. While these works optimize agent workloads from different perspectives, they do not address inefficiencies arising from context engineering, which is the focus of our work.

8 Conclusion

In this paper, we identify context transformation overhead as a key bottleneck in LLM-based agent systems, arising from KV

cache invalidation and recomputation under common context engineering strategies. By observing that these transformations are segment-decomposable, we enable their execution ahead of time. We propose a lookahead programming model that expresses transformations as asynchronous operations, together with a runtime and scheduling support to execute them proactively without impacting latency-critical requests. Our results show that this approach effectively eliminates transformation overhead and significantly improves latency.

References

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. Infercept: Efficient intercept support for augmented large language model inference. *arXiv preprint arXiv:2402.01869*, 2024.

[2] 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.

[3] Anthropic. Building effective agents. https://www.anthropic.com/engineering/building-effective-agents, 2024.

[4] Anthropic. Claude code. https://www.anthropic.com/claude-code, 2025.

[5] Anthropic. Effective context engineering for ai agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents, 2025.

[6] Anthropic. How we built our multi-agent research system. https://www.anthropic.com/engineering/multi-agent-research-system, 2025.

[7] Qiaoling Chen, Zhisheng Ye, Tian Tang, Peng Sun, Boyu Tian, Guoteng Wang, Shenggui Li, Yonggang Wen, Zhenhua Han, and Tianwei Zhang. Concur: High-throughput agentic batch inference of llm via congestion-based concurrency control. *arXiv preprint arXiv:2601.22705*, 2026.

[8] Xinhao Cheng, Zhihao Zhang, Yu Zhou, Jianan Ji, Jinchen Jiang, Zepeng Zhao, Ziruo Xiao, Zihao Ye, Yingyi Huang, Ruihang Lai, Hongyi Jin, Bohan Hou, Mengdi Wu, Yixin Dong, Anthony Yip, Zihao Ye, Songting Wang, Wenqin Yang, Xupeng Miao, Tianqi Chen, and Zhihao Jia. Mirage persistent kernel: A compiler and runtime for mega-kernelizing tensor programs. *arXiv preprint arXiv:2512.22219*, 2025.

[9] Chester Curme and Mason Daugherty. Context management for deep agents. https://blog.langchain.com/context-management-for-deepagents, 2026.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[11] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You,

Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

[12] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2(3):4, 2024.

[13] Hugging Face. Open-source deepresearch - freeing our search agents. https://huggingface.co/blog/open-deep-research, 2025.

[14] Taosong Fang, Zhen Zheng, Zhengzhao Ma, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. Flashagents: Accelerating multi-agent llm systems via streaming prefill overlap. *Proceedings of Machine Learning and Systems*, 2026.

[15] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. {Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, 2024.

[16] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

[17] GLM-5-Team, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

[18] Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. Qoserve: Breaking the silos of llm inference serving. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1492–1507, 2026.

[19] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. Deepspeed-fastgen: High-throughput text generation for llms via mii and

deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[20] Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts llm performance. Technical report, Chroma, July 2025.

[21] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 3(4):6, 2023.

[22] Zhengding Hu, Vibha Murthy, Zaifeng Pan, Wanlu Li, Xiaoyi Fang, Yufei Ding, and Yuke Wang. Hedrarag: Co-optimizing generation and retrieval for heterogeneous rag workflows. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 623–638, 2025.

[23] Yichao Ji. Context engineering for AI agents: Lessons from building manus. https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus, 2025.

[24] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems*, 44(1):1–27, 2025.

[25] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 897–912, 2025.

[26] Hao Kang, Ziyang Li, Xinyu Yang, Weili Xu, Yinfang Chen, Junxiong Wang, Beidi Chen, Tushar Krishna, Chenfeng Xu, and Simran Arora. Thunderagent: A simple, fast and program-aware agentic inference system. *arXiv preprint arXiv:2602.13692*, 2026.

[27] 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.

[28] LangChain. Context engineering. https://blog.langchain.com/context-engineering-for-agents, 2025.

[29] LangChain. LangChain: The agent engineering platform. https://www.langchain.com, 2026.

[30] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 6342–6353, 2023.

[31] Bill Yuchen Lin, Yicheng Fu, Karina Yang, Faeze Brahman, Shiyu Huang, Chandra Bhagavatula, Prithviraj Ammanabrolu, Yejin Choi, and Xiang Ren. Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks. *Advances in Neural Information Processing Systems*, 36:23813–23825, 2023.

[32] 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)*, pages 929–945, 2024.

[33] Chien-Yu Lin, Keisuke Kamahori, Yiyu Liu, Xiaoxiang Shi, Madhav Kashyap, Yile Gu, Rulin Shao, Zihao Ye, Kan Zhu, Rohan Kadekodi, Stephanie Wang, Arvind Krishnamurthy, Luis Ceze, and Baris Kasikci. Telerag: Efficient retrieval-augmented generation inference with lookahead retrieval. *arXiv preprint arXiv:2502.20969*, 2025.

[34] Jerry Liu. LlamaIndex. https://github.com/jerryjliu/llama_index, 2022.

[35] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

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

[37] Microsoft. Autogen: Open-source framework for agentic ai. https://www.microsoft.com/en-us/research/project/autogen, 2026.

[38] MiniMax. Mini Agent. https://github.com/MiniMax-AI/Mini-Agent, 2026.

[39] NVIDIA. NVIDIA Inference Xfer Library (NIXL). https://github.com/ai-dynamo/nixl, 2026.

[40] Zaifeng Pan, Yitong Ding, Yue Guan, Zheng Wang, Zhongkai Yu, Xulong Tang, Yida Wang, and Yufei Ding. Fasttree: Optimizing attention kernel and runtime for tree-structured llm inference. In *Proceedings of Machine Learning and Systems*, 2025.

[41] Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. Kvflow: Efficient prefix caching for accelerating llm-based multi-agent workflows. *arXiv preprint arXiv:2507.07400*, 2025.

[42] Zaifeng Pan, Yipeng Shen, Zhengding Hu, Zhuang Wang, Aninda Manocha, Zheng Wang, Zhongkai Yu, Yue Guan, and Yufei Ding. Scalesim: Serving large-scale multi-agent simulation with invocation distance-based memory management. *arXiv preprint arXiv:2601.21473*, 2026.

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

[44] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[45] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

[46] Jiawei Ren, Yan Zhuang, Xiaokang Ye, Lingjun Mao, Xuhong He, Jianzhi Shen, Mrinaal Dogra, Yiming Liang, Ruixuan Zhang, Tianai Yue, Yiqing Yang, Eric Liu, Ryan Wu, Kevin Benavente, Rajiv Mandya Nagaraju, Muhammad Faayez, Xiyan Zhang, Dhruv Vivek Sharma, Xianrui Zhong, Ziqiao Ma, Tianmin Shu, Zhiting Hu, and Lianhui Qin. Simworld: An open-ended realistic simulator for autonomous agents in physical and social worlds. *arXiv preprint arXiv:2512.01078*, 2025.

[47] Rya Sanovar, Srikant Bharadwaj, Renee St Amant, Victor Rühle, and Saravan Rajmohan. Leanattention: Hardware-aware scalable attention mechanism for the decode-phase of transformers. *Proceedings of Machine Learning and Systems*, 7, 2025.

[48] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[49] Peter Steinberger and OpenClaw Community. OpenClaw: Personal ai assistant. https://openclaw.ai, 2026.

[50] StepFun, Bin Wang, Bojun Wang, Changyi Wan, Guanzhe Huang, Hanpeng Hu, Haonan Jia, Hao Nie, Mingliang Li, Nuo Chen, Siyu Chen, Song Yuan, Wuxun Xie, Xiaoniu Song, Xing Chen, Xingping Yang, Xuelin Zhang, Yanbo Yu, Yaoyu Wang, Yibo Zhu, Yimin Jiang, Yu Zhou, Yuanwei Lu, Houyi Li, Jingcheng Hu, Ka Man Lo, Ailin Huang, Binxing Jiao, Bo Li, Boyu Chen, Changxin Miao, Chang Lou, Chen Hu, Chen Xu, Chenfeng Yu, Chengyuan Yao, Daokuan Lv, Dapeng Shi, Deshan Sun, Ding Huang, Dingyuan Hu, Dongqing Pang, Enle Liu, Fajie Zhang, Fanqi Wan, Gulin Yan, Han Zhang, Han Zhou, Hanghao Wu, Hangyu Guo, Hanqi Chen, Hanshan Zhang, Hao Wu, Haocheng Zhang, Haolong Yan, Haoran Lv, Haoran Wei, Hebin Zhou, Heng Wang, Heng Wang, Hongxin Li, Hongyu Zhou, Hongyuan Wang, Huiyong Guo, Jia Wang, Jiahao Gong, Jialing Xie, Jian Zhou, Jianjian Sun, Jiaoren Wu, Jiaran Zhang, Jiayu Liu, Jie Cheng, Jie Luo, Jie Yan, Jie Yang, Jieyi Hou, Jinguang Zhang, Jinlan Cao, Jisheng Yin, Junfeng Liu, Junhao Huang, Junzhe Lin, Kaijun Tan, Kaixiang Li, Kang An, Kangheng Lin, Kenkun Liu, Lei Yang, Liang Zhao, Liangyu Chen, Lieyu Shi, Liguo Tan, Lin Lin, Lin Zhang, Lina Chen, Liwen Huang, Liying Shi, Longlong Gu, Mei Chen, Mengqiang Ren, Ming Li, Mingzhe Chen, Na Wang, Nan Wu, Qi Han, Qian Zhao, Qiang Zhang, Qianni Liu, Qiaohui Chen, Qiling Wu, Qinglin He, Qinyuan Tan, Qiufeng Wang, Qiuping Wu, Qiuyan Liang, Quan Sun, Rui Li, Ruihang Miao, Ruosi Wan, Ruyan Guo, Shangwu Zhong, Shaoliang Pang, Shengjie Fan, Shijie Shang, Shilei Jiang, Shiliang Yang, Shiming Hao, Shuli Gao, Siming Huang, Siqi Liu, Tiancheng Cao, Tianhao Cheng, Tianhao Peng, Wang You, Wei Ji, Wen Sun, Wenjin Deng, Wenqing He, Wenzhen Zheng, Xi Chen, Xiangwen Kong, Xianzhen Luo, Xiaobo Yang, Xiaojia Liu, Xiaoxiao Ren, Xin Han, Xin Li, Xin Wu, Xu Zhao, Yanan Wei, Yang Li, Yangguang Li, Yangshijie Xu, Yanming Xu, Yaqiang Shi, Yeqing Shen, Yi Yang, Yifei Yang, Yifeng Gong, Yihan Chen, Yijing Yang, Yinmin Zhang, Yizhuang Zhou, Yuanhao Ding, Yuantao Fan, Yuanzhen Yang, Yuchu Luo, Yue Peng, Yufan Lu, Yuhang Deng, Yuhe Yin, Yujie Liu, Yukun Chen, Yuling Zhao, Yun Mou, Yunlong Li, Yunzhou Ju, Yusheng Li, Yuxiang Yang, Yuxiang Zhang, Yuyang Chen, Zejia Weng, Zhe Xie, Zheng Ge, Zheng Gong, Zhenyi Lu, Zhewei Huang, Zhichao Chang, Zhiguo Huang, Zhirui Wang, Zidong Yang, Zili Wang, Ziqi Wang, Zixin Zhang, Binxing Jiao, Daxin Jiang,

Heung-Yeung Shum, and Xiangyu Zhang. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding. *arXiv preprint arXiv:2507.19427*, 2025.

[51] 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*, pages 1302–1316, 2025.

[52] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[53] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In *ICLR*, 2025.

[54] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A {Multi-Level} superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 21–38, 2025.

[55] Zhiqiang Xie, Hao Kang, Ying Sheng, Tushar Krishna, Kayvon Fatahalian, and Christos Kozyrakis. Ai metropolis: Scaling large language model-based multi-agent simulation with out-of-order execution. *Proceedings of Machine Learning and Systems*, 7, 2025.

[56] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[57] 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.

[58] 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*, pages 94–109, 2025.

[59] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

[60] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[61] 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. In *Eighth Conference on Machine Learning and Systems*, 2025.

[62] 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.

[63] Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 144–158, 2025.

[64] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

[65] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. Batchllm: Optimizing large batched llm inference with global prefix sharing and throughput-oriented token batching. *arXiv preprint arXiv:2412.03594*, 2024.

[66] 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.

[67] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, et al. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism. *arXiv preprint arXiv:2504.02263*, 2025.

[68] Yan Zhuang, Jiawei Ren, Xiaokang Ye, Jianzhi Shen, Ruixuan Zhang, Tianai Yue, Muhammad Faayez, Xuhong He, Ziqiao Ma, Lianhui Qin, Zhiting Hu, and Tianmin Shu. Simworld-robotics: Synthesizing photorealistic and dynamic urban environments for multimodal robot navigation and collaboration. *arXiv preprint arXiv:2512.10046*, 2025.