

Sutradhara: An Intelligent Orchestrator-Engine Co-design for Tool-based Agentic Inference

Anish Biswas
Microsoft Research
Bengaluru, India
anish2kb@proton.me

Kanishk Goel
Microsoft Research
Bengaluru, India
k.goel2601@gmail.com

Srivarshinee S
M365 Research
Bengaluru, India
varshinms2159@gmail.com

Jayashree Mohan
Microsoft Research
Bengaluru, India
jamohan@microsoft.com

Alind Khare
M365 Research
Bengaluru, India
alindkhare@microsoft.com

Anjaly Parayil
M365 Research
Bengaluru, India
aparayil@microsoft.com

Chetan Bansal
M365 Research
Redmond, USA
chetanb@microsoft.com

Ram Ramjee
Microsoft Research
Bengaluru, India
ramjee@microsoft.com

Abstract

Agentic applications are LLMs that iteratively invoke external tools to accomplish complex tasks. Such tool-based agents are rapidly becoming the dominant paradigm for deploying language models in production. Unlike traditional single-turn inference, agentic workloads chain together multiple LLM calls and tool executions before producing a final response, creating a new performance bottleneck that manifests as increased latency in First Token Rendered (FTR) of the final answer. Through analysis of requests at production scale, we reveal three critical challenges: tool calls account for 30-85% of FTR latency, KV cache hit rates collapse despite substantial context reuse across iterations, and sequential orchestration wastes potential intra-request parallelism. These bottlenecks stem from a design gap in which orchestrators and LLM engines operate as decoupled black boxes, preventing cross-layer optimizations.

We present SUTRADHARA, a co-designed agentic inference system that integrates orchestration with LLM serving through a thin API enabling three optimizations: overlap tool execution with subsequent LLM prefill using tool-aware prompt splitting, streaming tool execution to dispatch tools incrementally during decode rather than waiting for complete output, and orchestrator-aware cache management that uses semantic hints to improve hit rates and reduce thrashing. Implemented on vLLM, SUTRADHARA improves

the throughput-latency trade-off in agentic systems, sustains up to 77% higher load at the same median FTR latency, or reduces median FTR latency by up to 15% at the same load while reducing end-to-end latency by up to 11% on A100 GPUs.

CCS Concepts: • Software and its engineering → Abstraction, modeling and modularity; Scheduling; Memory management.

Keywords: Large Language Models; Agentic Inference; Inference serving; Scheduling; Memory Management

ACM Reference Format:

Anish Biswas, Kanishk Goel, Srivarshinee S, Jayashree Mohan, Alind Khare, Anjaly Parayil, Chetan Bansal, and Ram Ramjee. 2027. Sutradhara: An Intelligent Orchestrator-Engine Co-design for Tool-based Agentic Inference. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27)*, April 11–15, 2027, Heraklion, Greece. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3845814.3850086>

1 Introduction

Large language models have rapidly evolved from research prototypes to production systems powering mission-critical applications. While early deployments focused on simple query-response patterns, modern LLM applications increasingly adopt agentic architectures—autonomous systems that iteratively invoke LLMs and external tools to accomplish complex tasks. These systems represent a fundamental shift in how we deploy LLMs. Rather than treating models as stateless oracles that answer isolated queries, agentic architectures enable LLMs to reason, plan, and interact with the external world through iterative tool use [30].



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3012-2/2027/04

<https://doi.org/10.1145/3845814.3850086>

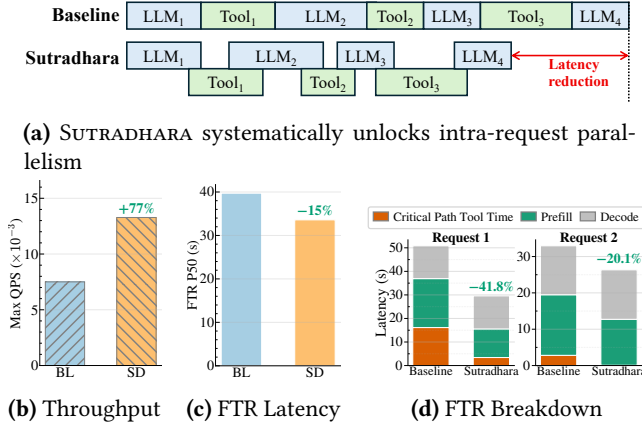


Figure 1. (a) SUTRADHARA systematically parallelizes the execution of LLM and tools and enables workload-aware KV eviction. (b) SUTRADHARA (SD) sustains upto 77% higher load than baseline (BL) at the same median FTR (38s) and (c) reduces median FTR by 15% at the same load. (d) For two random requests in the trace, these techniques reduce FTR by 20 – 42%.

However, this architectural shift introduces a critical performance challenge that existing LLM serving infrastructure fails to address: latency explosions in user-perceived response times. Traditional LLM serving systems optimize for Time-To-First-Token (TTFT) and inter-token latency, treating each inference request as an independent unit of work. In contrast, agentic applications chain together multiple LLM calls and tool invocations in iterative loops before producing a final response. The metric of relevance in this scenario is called First Token Rendered (FTR). It measures the elapsed time from submission of a user request until generation of the first token of the final user-visible answer. FTR latency encompasses not just a single LLM call, but the cumulative latency of multiple LLM calls interspersed with tool executions until the final response. In production systems, FTR latency can span tens of seconds, degrading user experience.

To understand these challenges systematically, we conduct the first large-scale empirical study of agentic inference performance using synthetic traces from production workloads in a large cloud provider. Our analysis reveals three critical insights that challenge conventional assumptions about LLM serving bottlenecks. (1) *Tool execution dominates tail latency*, accounting for 30-85% of FTR latency while individual tool calls can exceed LLM prefill time. While LLM prefill and decode have been the traditional focus of optimization efforts [1, 9, 12, 13, 17], we find that tool calls, often dismissed as "thin" external I/O operations—could be a major component of FTR latency. (2) *Sequential orchestration leaves parallelism unexploited*, as 50-80% of each iteration's prefill is tool-independent yet current systems wait for all tool outputs before beginning the next iteration; similarly tools are executed only after LLM decodes complete, while it is possible to stream tool executions as decode progresses.

(3) *KV cache thrashing destroys reuse opportunities* despite context reuse, as workload-agnostic LRU eviction thrashes shared prefixes when agentic requests execute concurrently.

These findings point to a fundamental architectural mismatch: **orchestrators and LLM engines operate as decoupled black boxes**, communicate only through opaque request-response interfaces. The orchestrator has knowledge of iterations, tool dependencies, and prompt composition, while the engine controls scheduling, batching, and KV cache management. Neither layer leverages information from the other to make globally optimal decisions. Breaking this abstraction barrier is essential to address bottlenecks.

We present SUTRADHARA, a co-designed agentic inference system that tightly integrates orchestration logic with LLM engine through a thin, principled API. SUTRADHARA enables three key optimizations: (1) parallel execution through prompt splitting, where iteration $i + 1$ performs a deterministic eager prefill of tool-independent context while iteration i 's tools execute, then incrementally extends the prefill when tool outputs arrive; (2) streaming tool dispatch during decode, where a streaming JSON parser identifies complete tool call objects as they are decoded and dispatches them immediately rather than waiting for full decode completion; and (3) orchestrator-aware cache management, where semantic metadata tags and reuse hints guide a priority-aware eviction policy that retains high-value blocks while evicting transient content, combined with request-aware scheduling that prioritizes completing in-flight requests. Figure 1 shows how intra-request parallelism enables SUTRADHARA to achieve significant latency reduction of up to 42% per request.

We implement SUTRADHARA as an asynchronous event-driven orchestrator built on top of vLLM v0.11.0, with targeted modifications to the v1 scheduler totaling fewer than 3500 lines of code. The orchestrator-engine interface consists of five new API calls (Table 1) that enable hint passing, parallel execution coordination, and cache metadata communication. Our implementation requires no changes to model architectures or inference kernels. We evaluate SUTRADHARA across A100-80GB GPUs using synthetic traces from production on Qwen3-14B. Compared to the vLLM baseline, SUTRADHARA achieves a better throughput-latency tradeoff, sustains up to 77% higher load at the same median FTR latency, or reduces median FTR latency by up to 15% at the same load while improving end-to-end latency by 11%. In summary, this paper makes the following contributions:

- **Empirical characterization.** Large-scale analysis of agentic workloads, identifying tool execution variance, sequentiality, and cache thrashing as sources of FTR latency (§3).
- **Co-designed architecture.** We propose SUTRADHARA, an orchestrator-engine co-design, with three novel techniques; prompt splitting with tool execution overlap, streaming tool dispatch, and semantic cache management that exploits agentic request structure (§4).

• **Implementation and evaluation.** We implement SUTRADHARA on vLLM with minimal invasiveness and demonstrate across models and workloads that it sustains up to 77% higher load at the same median FTR latency, or alternatively reduces median and tail FTR latency by up to 15% and 16% at the same load on an A100-GPU cluster (§5).

2 Background

2.1 Agentic Inference

Agentic inference represents a fundamental departure from traditional single-turn LLM inference. In standard LLM serving, a user submits a query, the model performs a single forward pass (prefill followed by autoregressive decode), and returns a complete response. The serving system optimizes for metrics like Time-To-First-Token (TTFT) and per-token decode latency, treating each request as an isolated, stateless computation. Agentic inference, by contrast, enables LLMs to interact with external tools and APIs through iterative execution loops. Rather than producing a final answer in one shot, the LLM reasons about a task, decides which tools to invoke, examines their outputs, and continues this process until reaching a satisfactory answer.

2.2 Agentic inference serving framework

Figure 2 illustrates the standard architecture for agentic inference serving, consisting of three primary components that interact through well-defined interfaces.

LLM Serving Engine. The engine handles low-level inference execution—batching requests, managing GPU memory, scheduling prefill and decode operations. Modern production systems use frameworks like vLLM [9] or SGLang [32], which implement optimizations such as continuous batching, PagedAttention, and prefix caching. The engine exposes a request-response API where clients submit prompts and receive generated token sequences. Importantly, the engine treats each LLM call as independent—it has no visibility into whether a request is part of an agentic workflow, or which iteration it belongs to.

Orchestrator. The orchestrator implements the control logic for agentic execution. Built as an asynchronous event-driven system (typically using Python’s `asyncio` or similar frameworks), it manages the iterative loop: constructing prompts from conversation history and tool outputs, dispatching LLM calls to the engine, parsing structured outputs to extract tool specifications, invoking tools through their respective APIs, and tracking iteration state. The orchestrator maintains semantic knowledge about request structure—which prompt sections depend on tool outputs, which tools are executing, and how context accumulates across iterations. However, this knowledge remains isolated within the orchestrator and is not communicated to the engine.

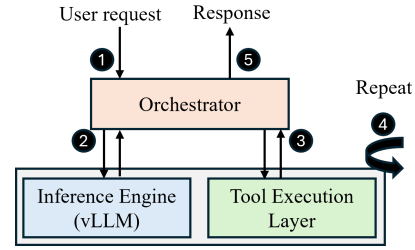


Figure 2. Workflow: (1) User request arrives; (2) Orchestrator sends LLM query and receives response; (3) Tools optionally invoked based on response; (4) Iterative loop of LLM and tools; (5) Final user-visible response returned.

Tool Execution Layer. Tools are services accessed through APIs, libraries, or sandboxed execution environments [24]. Common tools include web search, code execution, file system operations, and API calls to third-party services. Tools execute asynchronously and independently; when multiple tools are invoked in the same iteration, they run in parallel [8]. Tool execution time varies dramatically based on query complexity, external service load, and network conditions, ranging from milliseconds to seconds.

Multi-iteration execution structure. An agentic request consists of multiple iterations, where each iteration i follows a structured pattern:

1. **LLM Call:** The orchestrator submits a prompt to the LLM engine containing system instructions, conversation history, and (for $i > 1$) outputs from previous tool calls. The engine performs prefill to process the prompt, then generates tokens auto-regressively during decode. For the intermediate iterations, the decode output specifies which tools to invoke, typically formatted as structured JSON. For the final iteration, the decode output is the user-facing final response.
2. **Tool execution:** The orchestrator parses the tool specifications and dispatches each tool for execution. Each iteration can dispatch multiple tools, that may execute independently and finish at different times.

Sequential execution. Current agentic systems enforce a strict sequential execution between prefills, decodes, and tool calls per iteration. Our analysis shows that there is scope to enable intra-request parallelism across iterations, thereby optimizing request completion latencies.

3 Analysis

This section presents the agentic workload characterization that highlights bottlenecks in orchestration and KV cache behavior. We conclude with design requirements for an optimized system.

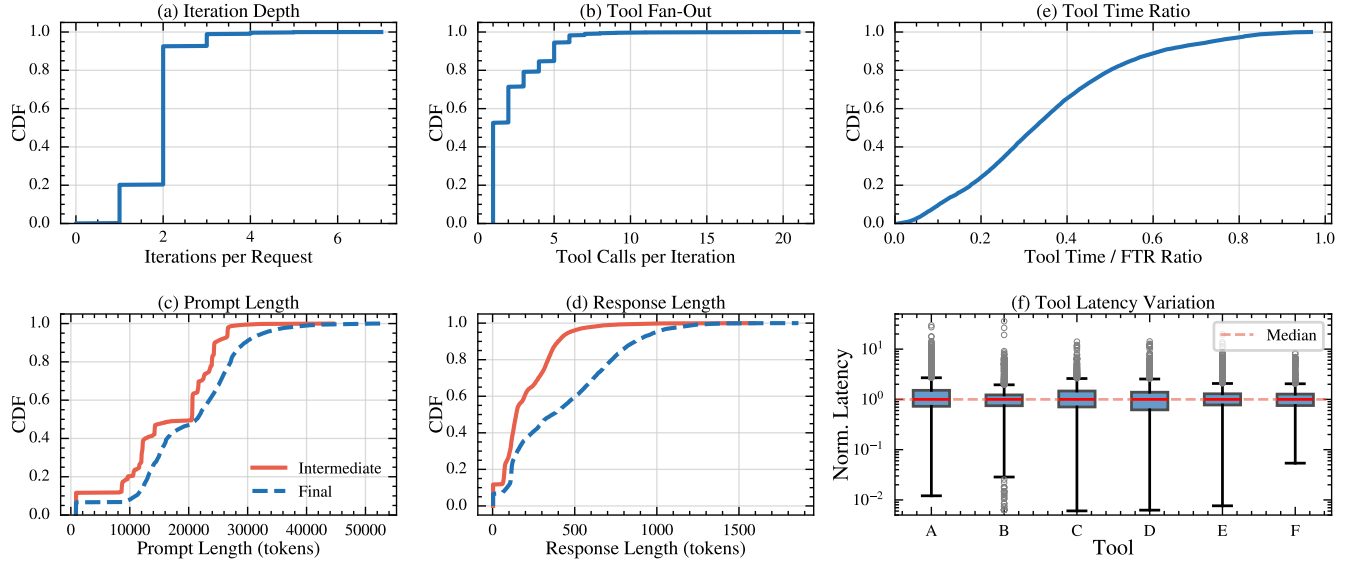


Figure 3. Detailed statistics of the agentic production trace. The distributions illustrate the structural characteristics of multi-step requests (a-d) and the associated tool execution dynamics (e-f). (f) denotes the normalized tool latency with respect to the median.

3.1 Trace collection

Agentic Platform. To understand the performance characteristics of agentic inference in production environments, we analyze workloads from a major cloud provider’s internal agentic platform deployed across thousands of enterprise customers. This platform provides a general-purpose orchestration framework that manages iterative LLM-tool execution loops. It dispatches requests to a heterogeneous fleet of LLM serving endpoints and coordinates calls to an extensible tool registry containing 20+ integrated services including web search APIs, enterprise chat, email, file searches, code execution, internal knowledge bases, and third-party SaaS integrations. The orchestrator implements a standard agentic execution pattern: issue an LLM request with available tools, parse structured output to identify tool calls, execute tools and collect outputs, append results to context, and iterate until the LLM produces a final response. The orchestrator could modify instructions and system prompts in any iteration depending on the set of tool calls made. This architecture is representative of popular agentic systems including LangChain [11], AutoGen [18], and proprietary enterprise frameworks.

Workload Generation. We analyze workloads issued on this platform with synthetic user profiles and data, instead of the real customer-facing queries as its prompt tokens are eyes-off for privacy reasons. The synthetic queries on this actual production platform are created through an automated pipeline that first provisions test users along with their synthetically generated enterprise context, including files, chats, and calendar events. Then, structured sets of

evaluation queries are executed on the platform, each designed to replicate realistic tasks such as document retrieval, summarization, and information search. By modeling workflows and entity relationships, the synthetic workload environment closely mirrors production scenarios. We also validate all the observations gained from synthetic workloads against customer-facing production workload. We analyze 6000 agentic requests across different user categories.

3.1.1 Metrics and Definitions.

First Token Rendered (FTR). FTR is defined as the elapsed time from user request submission to the rendering of the *first token* of the final user-visible response.

Iteration Depth. No. of LLM calls in an agentic request.

Tool Call Fan-Out. Number of tool invocations *per iteration*.

3.2 Trace statistics

We first present the overall trace characteristics. For each request, we categorize the number of LLM iterations into two types; intermediate iterations that perform tool invocations and a final iteration that generates the user visible response and doesn’t result in any tool calls. Figure 3 presents the overall trace characteristics (a) the distribution of iteration depth per request (b) tool call fan out per iteration, (c) prefill length distribution and (d) generated token distribution by iteration type. A median agentic request issues about 2 LLM iterations, one intermediate and one final, while in the tail, the iteration depth goes up to 7. Similarly, the per iteration tool call fan out is 2 in the median case, but some iterations could have as high as 21 tool calls made in a single iteration. The median prompt length across intermediate and final iterations is about 20K tokens, while the tail could be 3x

higher. The majority of the context is the system prompt (that could vary depending on the iteration and tools invoked in the previous iteration), and tool-specific instructions and formatting metadata. However, the median generated tokens for intermediate iterations is about 5x lower than the final iteration; this makes the intermediate iterations more prefill-bound, while the final iteration is decode-bound.

3.3 Trace-driven analysis

We present our key findings on the production workload.

Finding 1 : Tool call execution dominates the tail FTR. Figure 3(e) presents the cumulative distribution of tool execution time as a fraction of total FTR latency across our workload. While the median request spends only 32% of its FTR latency executing tools (with the remaining 68% spent in LLM prefill and decode), the tail exhibits dramatically different behavior. At the 90th percentile, tool execution accounts for 61% of FTR latency, climbing to 85% at the 99th percentile. This tail dominance contradicts the conventional assumption that tool calls are lightweight I/O operations contributing marginally to end-to-end latency [12, 17].

Our analysis further reveals variability in tool execution latency. Figure 3(f) shows box plots of tool latencies normalized to each tool’s median (p50) for six representative production tools. Even after normalization, all tools exhibit wide dispersion and pronounced right tails. By p75, normalized latency reaches $1.23\text{--}1.52 \times \text{p50}$ across tools, and at p90 spans $1.60\text{--}3.28 \times \text{p50}$. This heavy-tailed behavior indicates that tool execution latency remains highly sensitive to factors such as query complexity, backend contention, and the volume of data accessed (e.g., in search-style tools). As a result, tool execution time remains inherently difficult to predict, making tool-execution-time-aware KV caching strategies impractical in practice [12].

The magnitude of tool execution time presents a clear optimization opportunity. Even partially overlapping tool execution with subsequent LLM computation could yield significant latency reductions. However, exploiting this opportunity requires tight coordination between the orchestrator and LLM engine, which is a capability fundamentally absent in current black-box architectures where components communicate solely through request-response interfaces.

Finding 2 : Sequential orchestration leaves substantial parallelism unexploited. Current orchestrators enforce strict sequential execution within each agentic request: iteration i completes LLM decode, all tool calls execute to completion, all tool outputs return, and only then does iteration $i + 1$ begin prefill. This pipeline unnecessarily serializes the three phases – prefill, decode, and tool execution that could potentially overlap. We identify two specific sources of unexploited parallelism that stem from this rigid sequencing.

Opportunity 1: Prefill-tool overlap. Figure 4 analyzes prompt composition across iterations, measuring the fraction of each

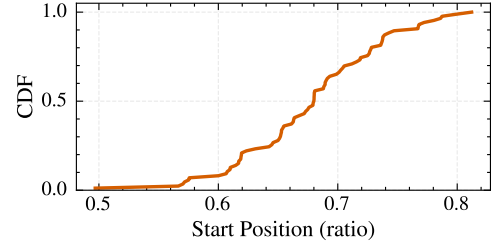


Figure 4. CDF of prompt prefix fraction independent of tool output

prompt consisting of tool-independent content versus tool-dependent outputs. We observe that 50–80% of iteration $i + 1$ ’s prompt is available when iteration i finishes decode. System instructions, conversation history, and templates don’t depend on tool outputs. Only the final 20–50% needs tool results. This creates a natural split point. The orchestrator could start prefill on the part of the prompt independent of tool outputs while tools execute, and extend the prefill incrementally.

Opportunity 2 : Decode-tool overlap. LLMs generate tool calls as JSON: [{"tool": "search", "query": "..."}, {"tool": "plot", "query": "..."}]. Current orchestrators wait for the entire array before dispatching tools. But once the first JSON object completes (closing `}`), that tool can execute immediately, allowing for streaming tool execution as decodes progress.

Finding 3 : KV cache thrashing lowers prefix reuse.

Agentic requests exhibit significant context reuse both across iterations within a request, as well as across requests that make similar tool calls. Across iterations, conversation history accumulates, whereas system prompts and instruction templates to consume tool output depend on tool call made in the prior iteration; such system prompts could be shared across requests that make similar tool calls. Standard prefix caching mechanisms should enable high cache hit rates by reusing these shared prefixes. However, we observe systematic cache thrashing that severely degrades performance under concurrent agentic workload execution.

Root cause: Workload-agnostic LRU eviction. Popular serving platforms [9, 32] employ Least Recently Used (LRU) eviction policies that treat all cached KV blocks identically, lacking awareness of agentic request structure. Agentic workloads systematically violate the temporal locality assumptions underlying recency-based eviction policies. Every tool invocation introduces a pause during which unrelated requests continue to consume cache capacity. Although the suspended request is highly likely to resume after tool completion, its KV state is no longer recent. Consequently, contexts with high future reuse probability are frequently evicted before the workflow resumes. Unlike conventional chat workloads where such delayed resumptions occur sporadically, agentic systems introduce them as a fundamental execution pattern.

To understand this thrashing behavior, consider three concurrent agentic requests R_1 , R_2 , and R_3 shown in Figure 5.

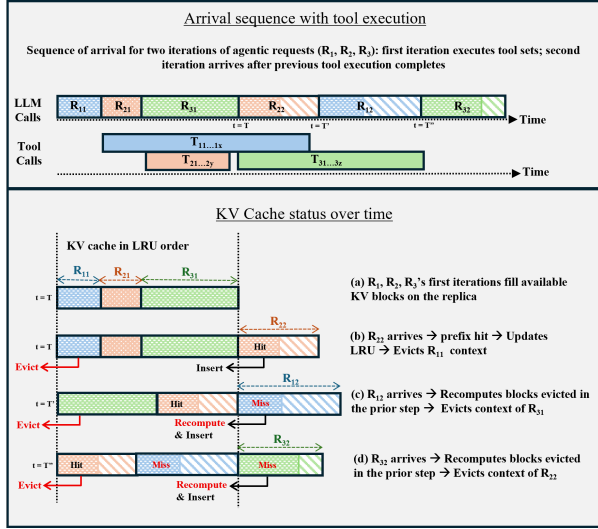


Figure 5. Threshing due to workload-agnostic KV eviction policy

Each request performs two iterations, where the first iteration generates tool calls (denoted $T_{11..1x}, T_{21..2y}, T_{31..3z}$ for requests R_1, R_2, R_3) and the second iteration can only begin after the previous iteration's tool execution completes.

At time $t = T$, all three requests complete their first LLM calls (R_{11}, R_{21}, R_{31}) and their KV blocks fill the available cache capacity in LRU order (Figure 5(a)). When R_2 's second iteration (R_{22}) arrives at $t = T'$, it gets a prefix cache hit from R_{21} . However, inserting R_{22} 's new KV blocks causes LRU to evict R_{11} 's context (Figure 5(b)). Subsequently, when R_{12} arrives at $t = T''$, it must recompute the evicted blocks from R_{11} , which in turn triggers eviction of R_{31} 's context (Figure 5(c)). This cascading pattern continues as R_{32} arrives at $t = T'''$ and must recompute R_{31} 's evicted blocks, further evicting R_{22} 's context (Figure 5(d)). The problem is that LRU evicts by recency, ignoring that R_{11}, R_{21}, R_{31} are first-iteration contexts that will be reused by blocked second iterations. We address this by using workload-aware eviction hints to evict KV-cache regions with lower reuse likelihood (§4.3).

In this work, we assume only HBM caching; thus, evicted KV blocks are recomputed. Alternatively, these blocks could be offloaded to secondary storage (e.g., CPU) using systems such as LMCACHE [4] or Mooncake [23]. In that case, our ideas remain applicable, as prefetching the right KV blocks is still required due to the HBM-storage latency hierarchy.

3.3.1 Summary. Our three key findings reveal a fundamental architectural problem: current agentic systems suffer from a **lack of co-design between the orchestrator and LLM inference engine**. Today's architectures treat these components as independent black boxes that communicate solely through opaque request-response interfaces. The orchestrator dispatches individual LLM calls and waits for complete responses before proceeding, while the engine treats each request as an isolated inference job with no awareness

API Call	Purpose
submit_partial_prefill()	Submit tool-independent prompt slice
extend_prefill()	Append tool outputs to pinned partial prefill context
register_streaming_callback()	Receive partial decode outputs token-by-token
tag_kv_blocks()	Annotate cached KV blocks with semantic hints (e.g., system_prompt, response)
set_reuse_priority()	Set priorities among KV blocks for pinning

Table 1. APIs that enable orchestrator-engine co-design.

of its role in a multi-iteration agentic workflow. This decoupling prevents both layers from exploiting information that could enable critical optimizations.

4 SUTRADHARA: Design and Implementation

SUTRADHARA extends the standard LLM serving architecture with a thin coordination layer that enables the orchestrator to communicate semantic hints about agentic request structure to the engine. As before, the orchestrator maintains knowledge of iteration boundaries, prompt composition, and tool dependencies, while the engine controls scheduling, batching, and KV cache management. However, through five new API calls shown in Table 1, the orchestrator guides engine decisions without requiring model-level modifications.

4.1 Parallel execution via deterministic eager prefill

Our analysis revealed that 50-80% of iteration $i+1$'s prompt is available immediately when iteration i completes decode, yet current systems wait for all tool outputs before beginning any prefill computation. This sequential execution leaves parallelism unexploited. However, naively starting prefill before tool outputs arrive introduces two challenges: (1) correctness: the engine must know where to splice tool outputs into the partial prompt, and (2) efficiency: the engine must retain the prefilled KV cache while tools execute, potentially without premature eviction.

Figure 6 illustrates our approach across three iterations of an agentic request. Each iteration consists of prefill (P_i) and decodes (D_i), followed by tool executions (T_{ij} for tool j in iteration i). Figure 6a is the baseline sequential execution done by systems today; iteration 1 completes decodes D_1 , all the tool executions T_{11}, T_{12} and then iteration 2 begins prefill P_2 . This pattern repeats for subsequent iterations. The LLM engine schedules iterations from other agentic requests in the idle time between tool calls to keep GPU occupied, creating long intra-request sequential chains of execution.

Prompt splitting as demonstrated in Figure 6b breaks this sequential dependency by partitioning prompts into tool-independent and tool-dependent slices:

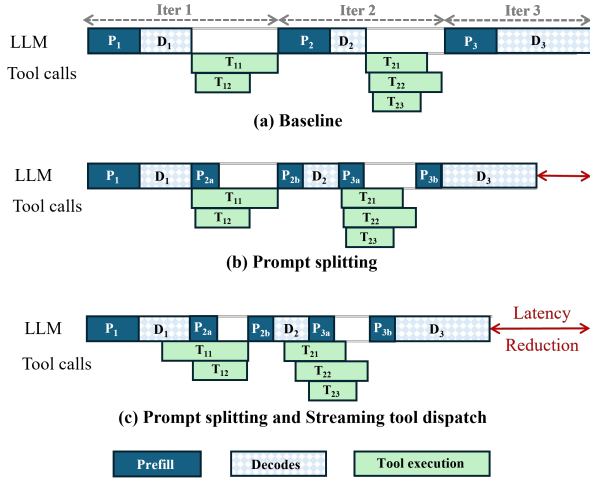


Figure 6. Intra-request parallel execution in SUTRADHARA

1. **Slice identification:** When the iteration i completes decodes and generates tool calls, the orchestrator aware of prompt template, identifies the insertion point where tool outputs will be spliced, typically between system instructions/history and the tool results section.
2. **Eager prefill execution:** The orchestrator submits the tool-independent prefix P_{2a} using `submit_partial_prefill()`, which returns a continuation handle. The engine computes prefill while tools execute concurrently. The KV cache blocks from partial prefill are tagged with high priority via `set_reuse_priority()` to prevent eviction.
3. **Prompt extension:** Once tool outputs arrive, the orchestrator constructs the tool-dependent suffix and calls `extend_prefill()` with the continuation handle. The engine splices the new content (P_{2b}) onto the pinned KV cache from P_{2a} , completes the prefill, and proceeds to decode D_2 . If tool execution fails or times out, the orchestrator discards the partial prefill and sets appropriate hints for its KV for the engine to release pinned resources.

This approach overlaps tool execution with a part of prefill, reducing the request's end-to-end latency.

4.1.1 Example. Consider an agent answering the query "Summarize the latest papers on KV cache management" as shown in Figure 7. During iteration i , the model emits the tool call `search_papers("KV cache")`. Once the tool call has been generated, the orchestrator transitions to a retrieval-specific prompt template for the next iteration. Note that this system prompt may vary from that of the previous iteration and hence cannot be reused directly from previous iteration's prefix cache. The prompt for iteration $i+1$ may therefore contain (1) an updated system prompt that instructs the model to reason over retrieved documents, (2) retrieval-specific instructions such as ranking and filtering criteria, (3) the conversation history and generated tool call, and finally (4)

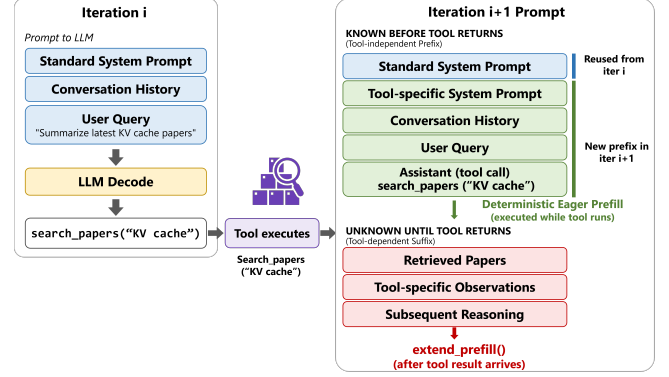


Figure 7. Eager prefill deterministically splits prefill-prompt and executes a part of the prefill in parallel with tool execution

the tool output itself. Before the search completes, components (1)–(3) are already known to the orchestrator, while only component (4) remains unavailable. SUTRADHARA therefore partitions the successor prompt into a tool-independent prefix and a tool-dependent suffix. While the search executes, it eagerly prefills the tool-independent region and retains the resulting KV state. When the search results arrive, the orchestrator inserts the retrieved documents and extends the prefill with only the remaining tool-dependent content. Importantly, SUTRADHARA never predicts the retrieved documents, future tool calls, tool arguments, or the agent's subsequent actions. The eager prefill operates exclusively on prompt content already known at tool-dispatch time. Consequently, SUTRADHARA incurs neither prediction errors nor rollback overheads; the only discardable work is a partial prefill whose associated tool fails or times out.

4.2 Streaming tool dispatch with decodes

Agentic systems generate tool calls in a structured representation such as JSON arrays during the LLM decode phase. Figure 6c shows that current systems wait for complete decode output before parsing and dispatching tools, introducing unnecessary serialization. However, once a complete tool invocation structure is decoded (e.g., the closing `}` of `"tool": "search", "query": "..."`), that tool can execute.

SUTRADHARA implements streaming tool dispatch through token-level callbacks from the engine to the orchestrator. When submitting a decode request that will generate tool calls (intermediate iterations), the orchestrator calls `register_streaming_callback()` with a handler function. The engine invokes this handler after each decoded token. The orchestrator maintains an incremental parser that accumulates tokens and identifies complete objects. In our implementation, the parser operates on JSON tool calls. When a tool call object closes (final `}` token), the parser extracts the tool name and parameters. As soon as a complete tool invocation is identified, the orchestrator dispatches it for execution without waiting for remaining decode tokens. Subsequent tool calls

in the JSON array are dispatched as they complete. The orchestrator tracks completion of all tools before proceeding to the next iteration.

Although our prototype processes OpenAI-style JSON structured tool calls, but the design is not tied to the structure. Streaming dispatch requires only an incrementally parseable representation that exposes tool-call boundaries before decode completion. Equivalent functionality can be implemented using XML or other structured formats.

4.3 Workload-aware KV cache management and scheduling

Our analysis identified systematic KV cache thrashing in concurrent agentic workloads despite significant context reuse. Figure 5 illustrates the root cause: workload-agnostic LRU eviction treats all cached blocks identically, causing cascading evictions that destroy reuse opportunities. LRU evicts based solely on recency, ignoring workload structure. It cannot distinguish between low-value blocks (transient content unlikely to be reused) and high-value blocks (first-iteration contexts that will be reused by second iterations currently blocked on tool execution). SUTRADHARA uses semantic tagging and priority-based eviction to tackle this.

KV block tagging. When the orchestrator submits LLM requests, it tags KV cache blocks with semantic metadata using `tag_kv_blocks()`.

- **System prompts:** Marked as *SYSTEM_PROMPT* with high reuse priority. Depending on the iteration type (intermediate or final) and the combination of tool calls made in the previous iteration, there could be many variants of system prompts that the orchestrator constructs using predefined rules; for e.g., the system prompt could contain instructions on how the output of the tool call made in the previous iteration should be consumed and interpreted. Such system prompts occur before the user-specific query and hence could be shared across many requests.
- **User specific query:** Marked as *USER_QUERY* and contains the request-specific context including user information which can result in prefix matches intra-request alone; across iterations of the same agentic request.
- **Tool outputs:** tagged as *TOOL_OUTPUT_ITER_i*, these are reused by subsequent intermediate iterations in an agentic request, provided that these iterations share the system prompt. If the system prompt in the next iteration changes due to a new tool call in the prior iteration, the prefilled tool output context from a prior iteration becomes useless as it is appended after the system prompt and user query.
- **Final response:** Tagged as *RESPONSE*. The final LLM call's decodes are the user-facing output tokens; these have no reuse potential and are candidates for eviction.
- **Partial prefills:** These are KV blocks from `submit_partial_prefill()`, tagged as *PARTIAL_PREFILL* with maximum priority until its extension prompt is completed.

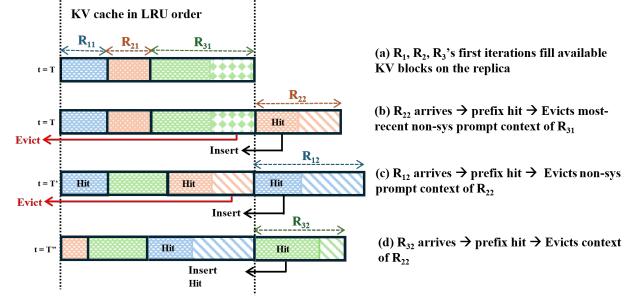


Figure 8. Workload-aware KV eviction policy

Priority-based eviction policy. Using the tags on the KV blocks, the engine's scheduler implements a priority-aware eviction policy that respects orchestrator hints. When cache capacity is exceeded, the scheduler evicts the lowest-priority blocks first, with LRU ordering as a tiebreaker.

Different prompt components exhibit different reuse characteristics across iterations. System prompts and active workflow state typically appear in most subsequent iterations, while tool outputs are often consumed by only a subset of downstream iterations. User-visible decode state generally has the lowest expected reuse. To reflect these differences, SUTRADHARA assigns eviction priorities to prompt segments based on their anticipated future reuse. The complete eviction priority used by SUTRADHARA (evicted first) $\rightarrow$ (evicted last) is as follows:

RESPONSE $\rightarrow$ TOOL_OUTPUT $\rightarrow$ USER_QUERY
 $\rightarrow$ SYSTEM_PROMPT $\rightarrow$ PARTIAL_PREFILL

Example. Returning to the scenario in Figure 5, our policy prevents cascading evictions as shown in Figure 8. At time $t = T'$, when R_{22} needs to insert new blocks, the eviction policy targets low-priority blocks (tool outputs) rather than R_{11} 's context, which is tagged with high reuse priority since R_{11} 's tools are executing. When R_{12} arrives at $t = T''$, it finds R_{11} 's blocks still cached, achieving a prefix hit and avoiding recomputation. Similarly, R_{31} survives until R_{32} arrives, preventing the cache miss chain.

Workload-aware scheduling. While scheduling is orthogonal to SUTRADHARA's optimizations, it remains a critical component for efficient agentic systems. Although agentic requests typically issue multiple LLM calls, the existing serving engines schedule at the granularity of individual LLM calls (e.g., FIFO), without global agentic context. As a result, requests that issue LLM calls more frequently may be unfairly prioritized over earlier-arriving agentic requests. To address this, SUTRADHARA enforces a workload-aware policy in the serving engines that preserves a global FIFO ordering of LLM calls with respect to the arrival time of agentic requests in the orchestration, ensuring fairness. Integrating more sophisticated scheduling techniques to prevent starvation, such as Autellix[17], is left for future work.

4.4 Implementation

We implement SUTRADHARA using $\approx 3,500$ lines of Python consisting of a serving engine layer and an independent orchestrator. We build the serving engine on top of vLLM (v0.11.0) and enable chunked prefill with prefix caching by default. For the orchestrator, we developed a lightweight, asyncio-based event-driven framework designed to accurately replay synthetic production traces. SUTRADHARA's modular design readily supports integration with alternative serving backends [2, 20]. Our engine modifications natively embed workload semantics, allowing the orchestrator to leverage this contextual metadata. For robustness, the components rely on a heartbeat-based membership protocol.

5 Evaluation

Our evaluation answers the following questions:

1. Does SUTRADHARA achieve a better throughput-latency trade-off for FTR and E2E latency on production agentic workloads?
2. Can SUTRADHARA generalize to disaggregated serving and diverse LLMs?
3. How does SUTRADHARA perform on open source agentic datasets?
4. What is the independent contribution of each optimization in SUTRADHARA?
5. Does SUTRADHARA increase KV cache hit rates?

5.1 Experimental Setup

Models. We evaluate SUTRADHARA on Qwen3-14B [29], a 14B-parameter open-source model featuring native function-calling support. At BF16 precision, this model fits within the memory budget of a single A100-80GB GPU supporting a 128K context window and accommodates the maximum prompt lengths observed in our workloads. SUTRADHARA's optimizations operate at the orchestrator-engine interface independent of the models or kernels used in the engine. To validate this, we evaluate SUTRADHARA on Gemma-3-27B in §5.3. We additionally evaluate SUTRADHARA on the Qwen3-30B model on a subset of a publicly available SWE-Bench trace.

Workloads. We evaluate SUTRADHARA across production and open-source traces.

Production Trace. It uses a subset of the production trace detailed in §3. We pick a stratified sample of 120 requests from the production trace, characterized by a high degree of tool-call fan-outs per request. We replay it using the original prompts, recorded tool outputs, and the decode lengths extracted from the trace. Tool-call latencies are normalized according to the observed tool-to-LLM ratio within the original traces as described in §3. Request arrival times follow a Poisson process. We report the mean success metrics with confidence bands across five seeds.

Open-Source Traces. The traces are derived from the following open-source benchmarks—

- **BFCL v4 Web Search [22]:** An open-source agentic benchmark that evaluates multi-hop web search via function-calling agents with standardized tools. We use 56 requests randomly sampled from an initial set of 100, each containing at least one multi-tool invocation. Agent trajectories are short, with a mean of 4.23 iterations and low tool fan-out per iteration (≈ 2). Tool execution dominates latency, averaging 1.09 s per call (variance 1.7 s) and accounting for approximately 40% of E2E request time.
- **SWE-Bench [7]:** An open-source agentic software engineering benchmark that captures long-horizon code exploration, execution, and repair over real-world GitHub issues. We replay 16 multi-turn agent trajectories generated by terminus-2 [5] with Qwen3-30B [29] on SWE-Bench Verified, selected from a pool of 1,497 trajectories. The trajectories for this subset are long-horizon (mean 10.9 iterations, up to 19) with high tool fan-out (mean 5.77 calls per tool-calling step, up to 14). Tool calls average 1.39 s (variance 6.61 s) and contribute 52–59% of overall execution time depending on load.

Both traces follow a strictly append-only prompt structure, where each iteration appends new context without modifying prior prompt prefixes. Most evaluations use the production trace, unless explicitly stated.

Hardware. We evaluate SUTRADHARA on A100-80GB GPUs [19]. For colocated experiments, we use a single A100. For disaggregated experiments, we use two A100-80GB GPUs - one dedicated prefill node and one dedicated decode node - connected over NVLink.

Baselines. We compare against a baseline using vLLM as the serving backend with Prefill-Decode co-location at a chunk size of 256, utilizing request-aware scheduling that maintains FIFO ordering at an agentic request level instead of individual LLM calls. SUTRADHARA builds on this baseline and additionally introduces prompt splitting, streaming dispatch, and orchestrator-aware KV cache management. We also compare against the PD disaggregated baseline in vLLM. Automatic prefix caching is enabled by default. We also compare against Continuum [12], a concurrent work that optimizes agentic systems by pinning KV cache blocks during tool execution using a time-to-live mechanism to prevent eviction-induced recomputation across iterations. We integrate Continuum with our orchestrator on a best-effort basis and set the TTL to 6s, based on mean tool call time in our production trace.

Metrics. We evaluate SUTRADHARA on two primary latency metrics. *First Token Rendered (FTR)* measures the time from the initial user request submission to the generation of the first token in the final user-visible response. *End-to-End (E2E)* latency reflects the total request completion time.

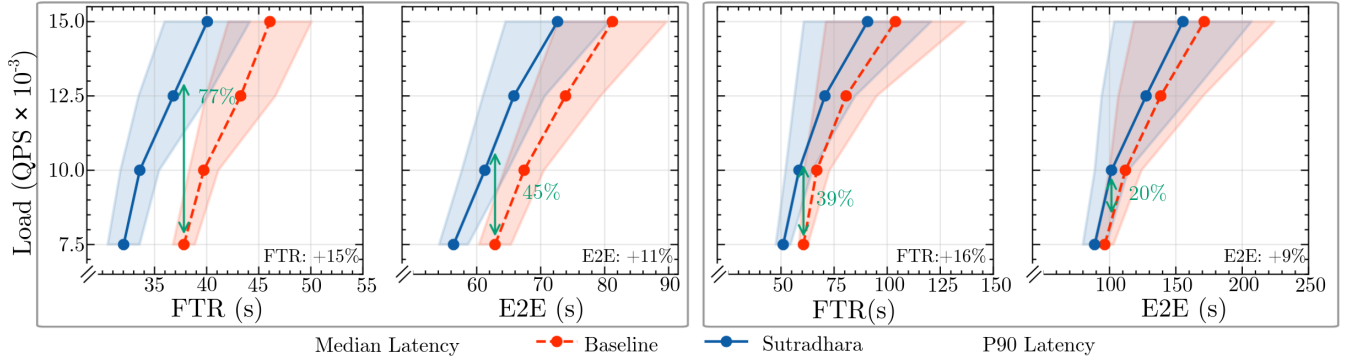
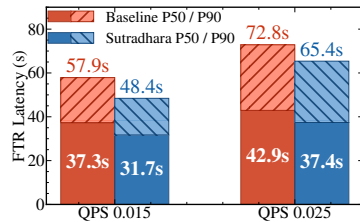
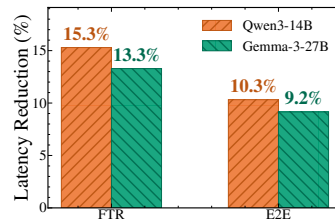


Figure 9. Serving capacity curves for SUTRADHARA and Baseline with p50/p90 FTR and E2E metrics on x-axis and ingest load (QPS) on y-axis. Curves that lie closer to the top-left are better: higher load sustained at lower latency. SUTRADHARA is closer to the top-left with a better throughput-latency tradeoff compared to baseline: sustains upto 77% higher load at the same latency or achieves upto 15% reduction in p50 FTR latency at the same load. Similar gains extend to E2E latency.



(a) Disaggregated serving



(b) Model generalizability

	QPS	BL (s)	SD (s)	Gain
BFCL v4	0.50	10.28	9.12	11.2%
	1.50	11.74	10.4	11.4%
	2.00	12.54	11.7	6.69%
SWE Bench	0.02	105.9	88.7	16.2%
	0.04	107.9	90.0	16.6%
	0.08	118.2	97.9	17.2%

(c) Median FTR on open source traces

Figure 10. SUTRADHARA latency gains generalize to (a) disaggregated serving, (b) different models (Gemma-3-27B), and (c) open source traces (BL = Baseline and SD = Sutradhara)

5.2 End-to-End Performance

We assess whether SUTRADHARA achieves a better throughput-latency tradeoff compared to baselines. Specifically, we evaluate SUTRADHARA on varying loads (0.0075, 0.01, 0.0125 and 0.015 QPS) and measure p50/p90 FTR and E2E latency. To characterize this trade-off, we plot serving capacity curves with ingest load (QPS) on the y-axis and latency on the x-axis. A system exhibits a better throughput-latency trade-off if it can sustain higher load at lower latency, or equivalently, achieve lower latency at the same load—corresponding to curves that lie closer to the top-left of the plot.

Figure 9 shows the serving capacity curves for both SUTRADHARA and baseline. SUTRADHARA achieves a better throughput-latency tradeoff compared to baseline, with its service capacity curve lying to the left of the baseline. At the same p50 FTR latency, SUTRADHARA sustains up to 77% higher load; conversely, for a fixed load, it achieves up to a 15% reduction in p50 FTR latency. SUTRADHARA also delivers improved trade-offs for tail latency. For the same p90 FTR latency, it sustains up to 39% higher load at the same latency, or achieves up to a 16% reduction in tail FTR latency at the same load. Similar benefits extend to end-to-end (E2E) latency, with up to a 9% reduction in p90 E2E latency. SUTRADHARA achieves a better trade-off as it unlocks intra-request

parallelism through its co-design and hides tool call time under prefills and decodes.

The gains are more pronounced for FTR than for E2E latency because the E2E metric includes the decode phase of the final iteration, which does not trigger any subsequent tool execution. As a result, this phase cannot benefit from prompt splitting or streaming dispatch optimizations, limiting the achievable acceleration in E2E latency.

5.3 Generalizability

Disaggregated Serving. We evaluate SUTRADHARA in disaggregated prefill-decode setting to assess whether SUTRADHARA generalizes to a different deployment. We run these workloads at 0.015 and 0.025 QPS, maintaining a per-GPU load equivalent to our collocated experiments. Figure 10 (a) summarizes the FTR latency improvements achieved under disaggregated serving. Across both evaluated load levels, SUTRADHARA consistently reduces both median and P90 FTR by 10.2% to 16%. These gains are consistent with the collocated results, demonstrating that SUTRADHARA’s optimizations transfer across deployment topologies without modification.

Model Generalizability. SUTRADHARA’s optimizations operate at the orchestrator-engine interface and universally apply to tool-calling LLMs served through vLLM. To validate

Table 2. Each optimization’s contribution at 0.0075 QPS. The median latency averaged over 3 seeds. PS: prompt splitting, DS: streaming dispatch, KV: cache management. Improvements are relative to Baseline+Sched.

Config	FTR (s)	E2E (s)	Cumulative		Incremental	
			FTR	E2E	FTR	E2E
Baseline	37.45	61.57	—	—	—	—
+PS	35.17	59.39	+6.1%	+3.5%	+6.1%	+3.5%
+PS+DS	32.07	56.03	+14.4%	+9.0%	+8.3%	+5.5%
+PS+DS+KV	31.39	54.93	+16.2%	+10.8%	+1.8%	+1.8%

this architecture-agnostic claim, we evaluate our system using Gemma-3-27B [26]: a more parameter-heavy model with a different attention mechanism than Qwen3-14B.

Figure 10 (b) shows the median FTR and E2E improvements for both models at 0.0075 QPS. SUTRADHARA reduces median FTR by 13.3% and E2E latency by 9.2% on Gemma-3-27B. This demonstrates SUTRADHARA’s latency improvements generalize across different models. The performance difference between the two models directly stems from Gemma-3-27B’s higher LLM compute requirements. Specifically, its prefill and decode phases are 1.34× and 1.60× slower, respectively. This increased compute time reduces the tool fraction of the overall FTR from 12.9% down to 9.0% limiting the available time window for SUTRADHARA’s optimizations to take effect.

Open Source Traces. We evaluate SUTRADHARA on the BFCL Web Search and SWE-Bench traces using the Qwen3-14B and Qwen3-30B models respectively. Figure 10 (c) shows the results across varying load levels. The baseline saturates at different load levels for both traces (2 QPS for BFCL and beyond the 0.08 QPS evaluated for SWE-Bench). SUTRADHARA’s optimizations generalize and reduce median FTR for both the traces. SUTRADHARA causes 11.4% and 17.2% p50 FTR reduction at 1.5 QPS load on BFCL, and 0.08 QPS load on SWE-Bench respectively. The lower median improvement observed on the open-source BFCL traces relative to the production trace is primarily due to two reasons; a) less tool call fan-out that limits the opportunities to overlap tool calls with the decode, and b) strictly append-only prompt structure that limits gains from prefill split as the splitted prefill may already remain prefix cached (from previous iteration’s prefill) leading to less overlap with tool calls.

5.4 Ablation Study

5.4.1 Cumulative Impact of Optimizations. Table 2 isolates the contribution of each optimization at 0.0075 QPS by incrementally adding them.

Prompt Splitting. Prompt splitting reduces median FTR by 6.1% and E2E latency by 3.5%. Although 50–80% of the prompt is available for the split (§3.3), these time savings are bounded by the extent of overlap between prefill and tools

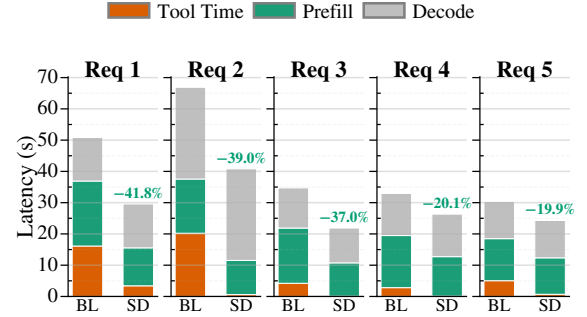


Figure 11. FTR Latency breakdown across five representative requests. BL is baseline, SD is SUTRADHARA

calls. For instance, whenever a tool executes faster than the partial prefill, the remaining prefill execution still remains on the critical path. Moreover, E2E latency gains remain less compared to FTR as decode dominates E2E latency and this optimization only performs prefill overlaps with tool calls.

Streaming Dispatch. Streaming dispatch further contributes to latency reduction by overlapping decodes with tool calls. It achieves an 8.3% FTR and 5.5% E2E latency improvement on top of prompt splitting, yielding cumulative improvements of 14.4% and 9.0%, respectively. Especially for high tool call fan-out requests, this optimization converts a purely sequential decode-then-dispatch pipeline into overlapped execution reducing the critical path.

KV Cache Management. This optimization further contributes a 1.8% improvement in both FTR and E2E latency. This reflects a direct reduction in prefill recomputation driven by higher cache hit rates under concurrent agentic workloads as demonstrated in §5.4.3. However, these increased hit rates don’t proportionally translate to similar latency gains as the decode dominates the E2E latency.

5.4.2 Per-Request FTR Latency Breakdown. To isolate the sources of performance improvement, Figure 11 decomposes the FTR latency of five randomly selected tool-heavy requests into three distinct phases: critical path tool time, total prefill time, and total decode time summed across intermediate iterations of each request. The breakdown reveals two key advantages. First, SUTRADHARA reduces the critical path tool time which is a direct consequence of streaming tool executions concurrently with the decode. Second, SUTRADHARA decreases the total prefill time compared to the baseline. This is due to higher cache hit rates (§5.4.3) and overlapping prefill execution with the tool invocations. The breakdown reveals how SUTRADHARA reduces FTR latency and enables better throughput-latency trade-offs (§5.2).

5.4.3 KV Cache Analysis. Figure 12 illustrates the decomposition of inter-request versus intra-request cache hits across varying iteration depths. At depth 0, cache hits are predominantly inter-request as requests share system prompt

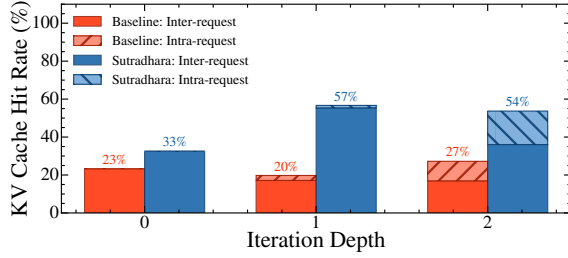


Figure 12. SUTRADHARA vs baseline on inter- vs intra-request cache hit rate by agentic requests’ iteration depth.

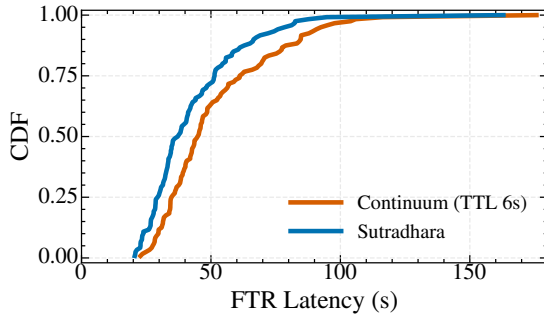


Figure 13. FTR-latency CDF for SUTRADHARA and Continuum.

prefixes. SUTRADHARA’s priority policy explicitly prevents these prefixes from eviction leading to higher cache hits. As iteration depth increases, context grows with accumulation of previous iterations’ tool call outputs. SUTRADHARA increases intra-request hits as partial prefills issued by SUTRADHARA contains tool call outputs from previous iterations and are given least eviction priority. Overall, SUTRADHARA improves the global cache hit rate from 21.8% to 44.6%.

6 Comparison with Concurrent Works

Figure 13 compares SUTRADHARA against Continuum [12] configured with TTL = 6s (average tool execution time in production trace). SUTRADHARA reduces median FTR by 17% over Continuum. Continuum addresses only the cache thrashing bottleneck and does not overlap tool execution with prefill computation nor dispatches tools during decode. The execution in Continuum remains sequential regardless of TTL configuration. Furthermore, TTL-based pinning is sensitive to tool execution variance. A higher variance leads to incorrect estimations of optimal TTL which negates benefits of KV pinning. Our trace shows a high variation in tool call times (§3), meaning a fixed TTL of 6s may cause cache evictions leading to higher tail latency. SUTRADHARA’s semantically aware priority based cache eviction policy remains effective regardless of tool execution variance.

7 Limitations and Discussion

Limitations of Orchestrator-Engine Coupling. SUTRADHARA’s co-design introduces a versioning dependency between the orchestrator and serving engine. While this coupling is necessary to enable streaming tool dispatch, prompt-splitting coordination, and workflow-aware cache management, it reduces the modularity of traditional black-box deployments. Specifically, SUTRADHARA’s extended five-API interface introduces a tight versioning dependency between the orchestrator and the LLM serving engine. Upgrading either component requires verifying API compatibility, adding operational overhead compared to traditional black-box deployments where the two layers evolve independently. Consequently, organizations adopting SUTRADHARA must coordinate releases across both layers rather than treating the inference engine as a modular, drop-in replacement.

Generality across agentic frameworks. SUTRADHARA’s overlap opportunity is fundamental to the ReAct pattern and is not a property of a specific harness. Every tool-based agent has three common elements: a tool-independent prompt prefix, a tool-execution wait period, and a well-defined LLM-tool boundary. SUTRADHARA only requires exposing this boundary through our five-call interface (Table 1); no model or kernel changes are required.

Frameworks like LangGraph [10] are well positioned to adopt SUTRADHARA’s interface. LangGraph’s graph-based execution model exposes explicit boundaries between LLMs and tool execution as discrete named nodes. The key adaptation is extending the LLM node to support concurrent operations during decoding by submitting tool-independent prompt content early and launching tool calls as soon as their specifications are generated. These changes are confined to the LLM node and the interface layer between the orchestrator and the engine, leaving LangGraph’s core execution model intact. We expect the same interface to generalize to other graph-based frameworks such as AutoGen [18] as they adopt similar execution models.

Multi-node deployment. Multi-node deployment is orthogonal to SUTRADHARA’s design because our optimizations operate at the per-node orchestrator-engine interface. Cross-node routing and cache-aware placement remain promising directions for future work.

8 Related Work

We now discuss literature relevant to SUTRADHARA. Table 3 compares SUTRADHARA with recent related works.

Agent Development. Several frameworks, such as LangGraph [10], AutoGen [18], DSPy [25], and Palimpsest [15], support agentic development by providing orchestration strategies for different applications. However, these frameworks treat the LLM backend as a black box and focus solely on orchestrator design. In contrast, our proposed interface

enables co-design between the orchestrator and the LLM backend, which can be leveraged by these frameworks to improve system efficiency.

Static Agent Workflow Optimization. Prior works have explored optimizing agentic inference, but they differ fundamentally from SUTRADHARA in scope and assumptions. Parrot [13] and Murakkab [3] expose agentic applications through declarative APIs that represent execution as static DAGs, enabling the runtime to optimize throughput–latency trade-offs using workflow structure. These approaches assume that the execution graph is known a priori. In contrast, the tool-based agents targeted by SUTRADHARA dynamically generate tool invocations, branch based on intermediate results, and may execute unpredictable tool loops.

Agentic Scheduling and Resource Management. Autelix [17] proposes a non-clairvoyant scheduler to improve throughput under agentic workloads, while Circinus [14] introduces SLO-aware query planning to improve goodput. These systems primarily treat agentic inference as a scheduling problem, optimizing how requests compete for LLM resources. However, our workload characterization shows that a substantial fraction of end-to-end latency arises outside the LLM itself due to tool execution delays and serialization across agent iterations. SUTRADHARA addresses this complementary bottleneck through orchestrator–engine co-design that exposes tool-level information to the serving system.

Cache Optimization for Agentic Workloads. Recent work has explored exploiting agent semantics to improve cache efficiency. KVFlow [21] leverages execution traces from static multi-agent workflows to improve prefix-cache reuse, while DroidSpeak [16] explores KV-cache sharing across collaborating agents. Continuum [12] seeks to improve cache utilization by predicting future tool invocation times. These approaches focus on cache reuse or predictive scheduling. In contrast, SUTRADHARA differs by targeting dynamic tool-based agents whose execution structure is not known a priori and whose control flow depends on tool outputs generated at runtime. Rather than relying on static workflow structure or prediction, SUTRADHARA exposes semantic reuse hints directly from the orchestrator and overlaps execution using information that is deterministically available at runtime.

Tool-Aware Execution Overlap. The closest work, Conveyor [28], overlaps partial tool execution with decoding. However, partial execution is infeasible for the agentic applications considered in the paper, as many tools cannot execute with incomplete parameters. Instead, SUTRADHARA tackles the practical challenge of handling tool-call fan-out, overlapping them with decoding.

Speculative tool execution. A distinct line of work accelerates agentic inference by speculating on future tool calls before the agent has decided which tool to invoke, and relies on additional machinery to recover when that speculation

Table 3. Comparison of related works against SUTRADHARA. ✓-supported; ✓-limited support; ✗-absent

Works	Dynamic Tool-Based Agents	Workload-Aware KV Cache Management	Tool-Decode Execution Overlap	Prefill-Tool Overlap
Parrot [13]	✗	✗	✗	✗
Murakkab [3]	✗	✗	✗	✗
KV Flow [21]	✗	✓	✗	✗
Autelix [17]	✓	✗	✗	✗
Continuum [12]	✓	✓	✗	✗
Conveyor [28]	✓	✗	✓	✗
Sutradhara (Ours)	✓	✓	✓	✓

is wrong. Speculative Actions [31] pre-executes predicted future tool calls in parallel branches using a smaller, faster model, wasting compute on discarded branches when the prediction is incorrect. PASTE [27] predicts future tool calls and infers arguments from offline-mined execution histories, requiring a pre-mined history corpus and a hardware slack budget to schedule the speculative work, and does not address KV-cache memory inefficiencies. Hooper et al. [6] launch tools early using speculation on partial streaming inputs and rely on fine-tuned models to cancel incorrect calls, which requires model fine-tuning and rollback logic. SUTRADHARA differs fundamentally: it operates only on information already determined by the workflow. Consequently, it incurs neither prediction errors nor rollback overhead and remains orthogonal to speculative tool execution approaches. Further, it introduces additional optimizations such as splitting prefill to reduce latency.

9 Conclusion

We present an extensive analysis of production tool-based LLM agentic applications, identifying three fundamental bottlenecks: (i) tool latency dominates end-to-end response time (30–85%), (ii) KV-cache effectiveness degrades despite significant context reuse across iterations, and (iii) sequential orchestration under-utilizes intra-request parallelism. These inefficiencies stem from decoupling orchestration from the LLM backend. We address this with SUTRADHARA, a co-designed agentic serving system. SUTRADHARA enables prompt-level parallelism, overlaps tool-call fan-outs with decoding, and incorporates workload-aware KV-cache management and scheduling. SUTRADHARA sustains up to 77% higher load at the same median latency, highlighting the need for coordinated orchestration–LLM co-design to scale agentic systems efficiently.

10 Acknowledgments

We thank our shepherd, along with the anonymous ASPLOS reviewers, for their valuable feedback.

References

- [1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10–12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 117–134. <https://www.usenix.org/conference/osdi24/presentation/agrawal>
- [2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed-Inference: Enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [3] Gohar Irfan Chaudhry, Esha Choukse, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. 2025. Murakkab: Resource-Efficient Agentic Workflow Orchestration in Cloud Platforms. arXiv:2508.18298 [cs.MA] <https://arxiv.org/abs/2508.18298>
- [4] Yihua Cheng, Yuhao Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. 2025. LMCACHE: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. CoRR abs/2510.09665 (2025). arXiv:2510.09665 doi:10.48550/ARXIV.2510.09665
- [5] Harbor Framework Team. 2026. *Harbor: A framework for evaluating and optimizing agents and models in container environments*. doi:10.5281/zenodo.20953922
- [6] Coleman Hooper, Minsuk Kang, Sehoon Moon, Nikolai Lee, Ernie Wen, John Wawrzynek, Michael W. Mahoney, Yakun Sophia Shao, Amir Gholami, and Kurt Keutzer. 2026. Speculative Interaction Agents: Building Real-Time Agents with Asynchronous I/O and Speculative Tool Calling. arXiv:2605.13360 [cs.AI] <https://arxiv.org/abs/2605.13360>
- [7] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=VTF8yNQm66>
- [8] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An LLM Compiler for Parallel Function Calling. In *International Conference on Machine Learning (ICML)*.
- [9] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23–26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. doi:10.1145/3600006.3613165
- [10] LangChain AI. 2025. GitHub - langchain-ai/langgraph: Build resilient language agents as graphs. — github.com. <https://github.com/langchain-ai/langgraph>. [Accessed 11-12-2025].
- [11] LangChain AI. 2025. LangChain — langchain.com. <https://www.langchain.com/>. [Accessed 11-12-2025].
- [12] Hanchen Li, Qiuyang Mang, Runyuan He, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. 2025. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. arXiv:2511.02230 [cs.OS] <https://arxiv.org/abs/2511.02230>
- [13] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA. <https://www.usenix.org/conference/osdi24/presentation/lin-chaofan>
- [14] Banruo Liu, Wei-Yu Lin, Minghao Fang, Yihan Jiang, and Fan Lai. 2025. Circinus: Efficient Query Planner for Compound ML Serving. arXiv:2504.16397 [cs.DB] <https://arxiv.org/abs/2504.16397>
- [15] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpsest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*.
- [16] Yuhao Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. 2025. DroidSpeak: KV Cache Sharing for Cross-LLM Communication and Multi-LLM Serving. arXiv:2411.02820 [cs.MA] <https://arxiv.org/abs/2411.02820>
- [17] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. 2025. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. arXiv:2502.13965 [cs.LG] <https://arxiv.org/abs/2502.13965>
- [18] Microsoft. 2025. GitHub - microsoft/autogen: A programming framework for agentic AI — github.com. <https://github.com/microsoft/autogen>. [Accessed 11-12-2025].
- [19] Microsoft Azure. 2024. Azure VM NDm-A100-v4 sizes series. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndma100v4-series>.
- [20] NVIDIA. 2024. NVIDIA TensorRT-LLM. <https://github.com/NVIDIA/TensorRT-LLM>.
- [21] Zaifeng Pan, Ajikumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. 2025. KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows. arXiv:2507.07400 [cs.DC] <https://arxiv.org/abs/2507.07400>
- [22] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs. arXiv preprint arXiv:2305.15334 (2023).
- [23] Ruoyu Qin, Zheming Li, Weiran He, Jiale Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. 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 2025, Santa Clara, CA, February 25–27, 2025*, Haryadi S. Gunawi and Vasily Tarasov (Eds.). USENIX Association, 155–170. <https://www.usenix.org/conference/fast25/presentation/qin>
- [24] Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: a survey. *Frontiers of Computer Science* 19, 8 (2025), 198343. doi:10.1007/s11704-024-40678-2
- [25] Stanford. 2025. GitHub - stanfordnlp/dspy: DSPy: The framework for programming—not prompting—language models — github.com. <https://github.com/stanfordnlp/dspy>. [Accessed 11-12-2025].
- [26] Gemma Team. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] <https://arxiv.org/abs/2503.19786>
- [27] Kaiqiang Xu, Fan Lai, Zhenghang Ren, Xin Jin, and Kai Chen. 2026. Parallelizing Tool Execution and LLM Generation for Low-Latency Agent Serving. arXiv:2603.18897 [cs.DC] <https://arxiv.org/abs/2603.18897>
- [28] Yechen Xu, Xinhao Kong, Tingjun Chen, and Danyang Zhuo. 2024. Conveyor: Efficient Tool-aware LLM Serving with Tool Partial Execution. <https://openreview.net/forum?id=A0VvDN4arV>
- [29] An Yang et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] <https://arxiv.org/abs/2505.09388>
- [30] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629 (2022).

- [31] Naimeng Ye, Arnab Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. 2026. Speculative Actions: A Lossless Framework for Faster AI Agents. In *The Fourteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=P0GOk5wslg>
- [32] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, Clark Barrett, and Ying Sheng. 2023. SGLang: Efficient Execution of Structured Language Model Programs. *arXiv preprint arXiv:2312.07104* (2023).

A Artifact Appendix

A.1 Abstract

SUTRADHARA is a co-designed agentic inference system that couples the agent orchestrator with the LLM serving engine through a thin orchestrator↔engine API. This artifact contains the orchestrator and trace-replay harness as an installable Python package, the serving-layer changes as a python patch against vLLM v0.11.0, the BFCL v4 agentic trace used in the evaluation, one-command sweep scripts, and an analysis notebook that prints the resulting latency table and plots. One A100 80 GB GPU suffices for all experiments except disaggregated serving, which needs two A100 80 GB GPUs on the same node. The single-GPU capacity sweep takes ≈40 minutes.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Tool-aware prompt splitting, streaming tool execution during decode, orchestrator-aware KV-cache eviction.
- **Program:** SUTRADHARA orchestrator and trace-replay harness, plus a patched vLLM v0.11.0 server.
- **Compilation:** None. The patch is in python, installed editable over the official precompiled v0.11.0 CUDA wheel.
- **Model:** Qwen3-14B [29] (bf16, TP1), via Hugging Face.
- **Data set:** BFCL v4 [29] web_search agentic trace (56 requests with recorded per-call tool latencies), shipped in the repo.
- **Run-time environment:** Linux x86-64, NVIDIA driver ≥570, pixi provides Python 3.12, CUDA 12.9, torch 2.8.0+cu129.
- **Hardware:** 1×A100 80 GB (tested on A100 80 GB PCIe); 2×A100 80 GB on one node for disaggregated serving.
- **Metrics:** FTR and E2E latency (p50, p90).
- **Output:** Per-run CSV metrics and logs under experiments/; notebook prints the latency table.
- **Experiments:** BFCL capacity sweep (6 QPS points); BFCL disaggregated sweep (7 QPS points).
- **Disk space required?:** ≈50 GB (15 GB environment, ≈29 GB model cache, 1.4 GB vLLM checkout).
- **Time to prepare workflow?:** 30–45 minutes.
- **Time to complete experiments?:** ≈40 minutes on one A100 for the capacity sweep.
- **Publicly available?:** Yes, <https://github.com/microsoft/sutradhara>.
- **Code/data licenses:** MIT; the serving layer ships as a patch against vLLM (Apache-2.0).
- **Workflow framework used?:** pixi; shell and Python scripts; Jupyter for analysis.

- **Archived (DOI)?:** <https://doi.org/10.5281/zenodo.23270949>

A.3 Description

A.3.1 How to access. Download the archived artifact from <https://doi.org/10.5281/zenodo.23270949>, or clone <https://github.com/microsoft/sutradhara> (≈5 MB, traces included).

A.3.2 Hardware dependencies. One NVIDIA A100 80 GB GPU (two on the same node for disaggregated serving). Roughly 50 GB of free disk.

A.3.3 Software dependencies. All versions are pinned in `pixi.lock`: Python 3.12, CUDA toolkit 12.9, torch 2.8.0+cu129, transformers ≥4.57, and the analysis stack. Only an NVIDIA driver and git are needed from the host; numactl is used for NUMA pinning if present. See `README.md`.

A.3.4 Data sets. `experiment_traces/bfcl_trace.json` (JSONL, one request per line): 56 agentic requests derived from BFCL v4’s `web_search` category. Each record carries the model’s tool-calling turns and the measured wall-clock latency of every tool call, which the harness replays as real delays, so tool time is reproduced without network access.

A.3.5 Models. Qwen3-14B in bf16, downloaded from Hugging Face on first use (set `HF_HOME` to control cache placement). Prompt formatting, KV hints, and decode-stream trigger indices are computed with the Qwen tokenizer, so the model should not be changed for reproduction.

A.4 Installation

```
curl -fsSL https://pixi.sh/install.sh | bash
pixi install
pixi run install-serving-layer
```

Verify:

```
pixi run python -c "import sutradhara.orchestrator"
pixi run python -c "from vllm.v1.core.\
kv_block_types import KVBlockType"
```

A.5 Experiment workflow

Most results in the paper use production traces, which cannot be released. We therefore provide scripts that reproduce the key experiments on the open-source BFCL trace, which is also part of the paper’s evaluation (Fig. 10(c)). All experiments below use the BFCL trace. On BFCL, SUTRADHARA runs streaming dispatch and KV-cache management (`--decode-stream` and `--workload-aware-cache`); prompt splitting is a no-op on this trace’s append-only ChatML prompts.

Capacity sweep.

```
BFCL_GPU=0 ./scripts/bfcl_capacity_sweep.sh
```

The script replays the BFCL trace at 0.5, 1, 1.5, 2, 3, and 4 QPS. For each QPS point, it runs both BASELINE and SUTRADHARA, for 12 runs in total. Each run starts a patched

vLLM server, replays the trace under Poisson arrivals, executes tool calls using their recorded latencies, and writes performance metrics to `experiments/`. Both BASELINE and SUTRADHARA use the same trace, request order, and arrival schedule. The script can be repeated to quantify run-to-run variability and compute aggregate metrics with error bars.

Disaggregated serving.

```
BFCL_GPUS="0 1" ./scripts/bfcl_disaggregated.sh
```

This script runs the disaggregated serving stack (first GPU prefill, second GPU decode) for both BASELINE and SUTRADHARA at 0.5, 1, 2, 3, 4, 6, and 8 QPS, for 14 runs in total; the QPS list can be edited in the script. It requires at least two A100 GPUs on the same node. Our platform connects the GPUs over NVLink; performance may vary on systems with a different interconnect.

Analysis. All expected results in the next section are produced by running all cells of the analysis notebook, which reads outputs from `experiments/`:

```
analysis/bfcl_paper_experiments.ipynb
```

The environment variable below selects latest (most recent run, point estimates only) or all (default; aggregates all available runs and computes error bars):

```
BFCL_RUN_DIR_MODE=all
```

A.6 Evaluation and expected results

The notebook compares BASELINE and SUTRADHARA at every swept load and reports the relative gain. Expected results on

the BFCL trace (“cf.” marks the production-trace figure each result corresponds to):

- **Capacity curves (cf. Fig. 9):** SUTRADHARA sustains 25–90% higher load at the same median FTR, depending on the latency target.
- **Median latency (Fig. 10(c)):** SUTRADHARA reduces median FTR and E2E latency at every QPS point, by 4–11% (FTR) and 4–10% (E2E) in our reference run on one A100 80 GB PCIe (at 1.5 QPS, p50 FTR falls from 11.74 s to 10.4 s).
- **Disaggregated serving (cf. Fig. 10(a)):** SUTRADHARA reduces median FTR and E2E latency by 6–8%.
- **Per-request FTR breakdown (cf. Fig. 11):** SUTRADHARA reduces individual request latencies by up to 40% through lower critical-path tool time and less prefill time from higher KV-cache hit rates.

A.7 Experiment customization

- Sweep points (load, memory, seed, request ordering): `scripts/bfcl_capacity_sweep.sh`, lines 13–29.
- Optimization flags: `--prefill-split`, `--decode-stream`, `--workload-aware-cache`, and `--track-eviction-types`.

Each optimization is an independent flag, so any point in the ablation space can be run directly; flags compose into the output directory tag. The archive also includes the SWE-bench trace and `scripts/swe_capacity_sweep.sh` (two GPUs, TP=2). See `scripts/README.md` for all options.