

Tokencake: A KV-Cache-centric Serving Framework for LLM-based Multi-Agent Applications

Zhuohang Bian
Peking University

Teng Ma
Independent Researcher

Feiyang Wu
Peking University

Youwei Zhuo
Peking University

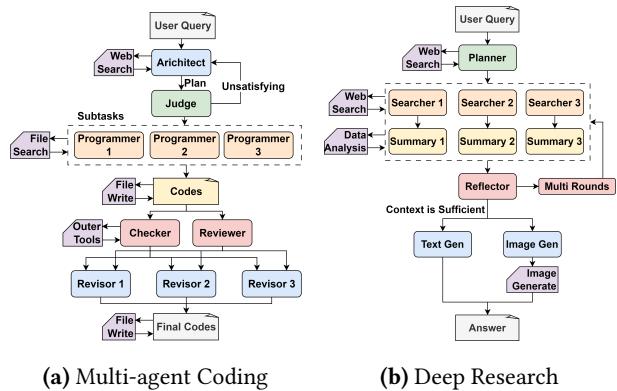
Abstract

Large Language Models (LLMs) are increasingly deployed in complex multi-agent applications that use external function calls. This workload creates severe performance challenges for the KV Cache: space contention leads to the eviction of critical agents' caches and time underutilization leaves the cache of agents stalled on long-running tool calls idling in GPU memory. We present Tokencake, a KV-Cache-centric serving framework that co-optimizes scheduling and memory management with an agent-aware design. Tokencake's Space Scheduler uses dynamic memory partitioning to shield critical agents from contention, while its Time Scheduler employs a proactive offload and predictive upload mechanism to repurpose GPU memory during function call stalls. Our evaluation on representative multi-agent benchmarks shows that Tokencake can reduce end-to-end latency by over 47.06%, improve effective GPU memory utilization by up to 16.9% compared to vLLM.

1 Introduction

Large Language Models (LLMs) are powerful reasoning engines, and applications built upon them are evolving from single-response generation to complex, multi-agent systems. This new approach has enabled powerful applications in domains like autonomous code generation[11], complex financial analysis[17], and realistic environment simulation[12]. The defining characteristic of these applications is a dual-interaction model: frequent external, **agent-tool** interaction and complex internal, **agent-agent** collaboration. Externally, these agents use function calls to interact with tools, data sources, and APIs. Internally, these systems orchestrate multiple specialized agents that collaborate to solve a larger problem.

Figure 1 illustrates this application model. For example, Code-Writer[19] and Deep-Research[15] are composed of internal pipelines of agents (e.g., programmers, reviewers, searchers) that in turn make frequent external calls to tools like file systems and web APIs. The combination of these complex internal dependencies and frequent, long-running external interactions results in workload patterns distinct from traditional LLM inference, introducing unique and significant performance challenges for the underlying serving infrastructure.



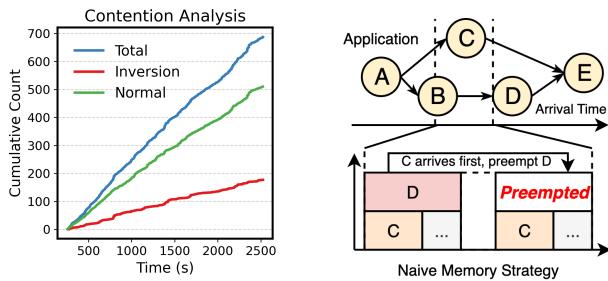
(a) Multi-agent Coding

(b) Deep Research

Figure 1. Example LLM-based Multi-Agent Applications. Each colored box represents a specialized agent. Purple boxes denote function calls to external tools.

These challenges are rooted in inefficient KV Cache management, which manifests as two critical problems: **time underutilization** and **space contention**.

First, time underutilization stems from the frequent and long-running function calls inherent to agentic workloads. An agent's execution follows an $LLM\ Inference1 \Rightarrow Function\ Call \Rightarrow LLM\ Inference2$ pattern, where its KV Cache sits unused during the function call. This forces a difficult trade-off: retain the cache and waste resources, or evict it and incur a costly recomputation. This inefficiency is substantial; as shown in Figure 3a, at peak moments, as much as 18.5% of the GPU KV Cache pool can be wasted by stalled agents, directly reducing the system's capacity for active requests, lowering the effective batch size, and degrading overall throughput. Systems like Teola [14] have identified the latency challenge posed by this pattern and propose a workflow-level optimization. Teola intelligently pipelines the execution of LLM and non-LLM micro stages, aiming to overlap the tool-use-time of one agent with the computational work of another. While this approach effectively hides latency by improving the application's end-to-end execution schedule, it is fundamentally compute-centric. Teola's scheduler optimizes the flow of operations but remains blind to the state of the underlying GPU memory resources. Consequently, the KV Cache of the stalled agent continues to idly



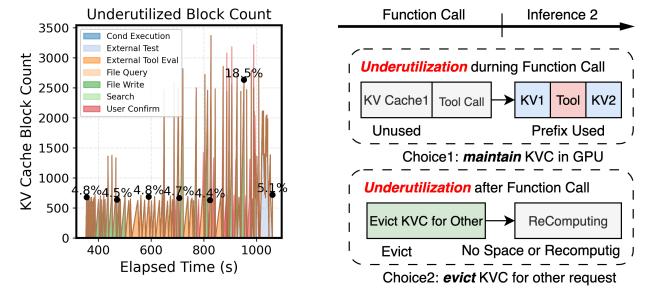
(a) Preemption events over time of the Code-Writer workload. (b) KV Cache blocks held by non-critical agents.

Figure 2. The Space Contention Problem.

occupy valuable GPU memory, a problem that is exacerbated when multiple agents stall concurrently.

Second, space contention arises as numerous agents compete for limited GPU memory. Agent-unaware memory allocation policies like FCFS often lead to performance bottlenecks, where a non-critical agent causes the eviction of a critical-path agent's KV Cache, a problem we refer to as critical inversion. We do an empirical measurement on the application Code-Writer, and count the preemption number of critical inversion. As shown in Figure 2a, these harmful preemptions happen frequently. This forces the evicted critical agent to undergo a costly context recomputation, stalling the entire application workflow. While systems like Parrot [8] and Autellix [10] are workflow-aware, their optimizations operate at the request level, agnostic to the fine-grained KV Cache contention between individual agents. It optimizes the order and batching of requests but does not manage the underlying memory allocation. Consequently, even with an optimal schedule, a high-throughput, non-critical task group identified by Parrot could still occupy GPU memory and inadvertently cause the eviction of a latency-sensitive, critical agent's KV Cache. This exposes a fundamental limitation: scheduling optimization alone cannot solve memory contention.

To address these challenges, we present Tokencake, a KV Cache-centric serving framework that co-optimizes scheduling and memory management through an agent-aware design. Tokencake begins with a frontend API that allows developers to define an application's internal agent collaborations and external tool interactions as a graph (Section 3.1). This graph enables two specialized schedulers to manage the KV Cache lifecycle with application-level context. To mitigate time underutilization, the **Time Scheduler** uses an event-driven, opportunistic policy to proactively offload the KV Cache of stalled agents during function calls and uses predictive uploading to hide data transfer latency (Section 4). To resolve space contention, the **Space Scheduler** employs a dynamic memory partitioning policy, guided by a hybrid



(a) Idle KV Cache blocks due to external function call. (b) The lifecycle of an agent's KV Cache during a function call.

Figure 3. The Time Underutilization Problem.

priority metric, to reserve memory for critical-path agents (Section 5). We further introduce optimizations such as CPU block buffering and gradual GPU block reservation to minimize the overhead of these frequent offload operations. Our evaluation shows that Tokencake significantly reduces end-to-end latency by over 47.06% compared to vLLM under high load and improves GPU memory utilization by over 16.9% (Section 7).

2 Background and Motivation

2.1 LLM-based Multi-Agent Applications

Agents. A common design pattern in modern agentic systems is the decomposition of complex problems into a series of sub-tasks handled by specialized agents [5, 6, 16]. These collaborations are often modeled as a Directed Acyclic Graph (DAG), where nodes represent agents and edges represent explicit dependencies. In these workflows, the dependencies are critically important; for instance, a reviewer agent cannot begin its task until a programmer agent has completed its code generation. Furthermore, not all agents contribute equally to the application's end-to-end latency. Some agents lie on the critical path, meaning any delay they experience directly increases the total time-to-result for the entire application.

Function Calls. A key feature of modern agents is the use of function calls to interact with external data sources or execute actions. These calls connect the LLM to a vast array of tools, such as database clients, code interpreters, or third-party web APIs. To standardize these interactions, the Model Context Protocol (MCP) [3] is emerging as an open standard, providing a unified interface and a rich ecosystem of pre-defined tools. As shown in Table 1, compiled from the official MCP documentation and our empirical measurements, the latencies of these common tools have a wide and often unpredictable distribution.

Table 1. Latency characteristics of common tools in MCP.

Tool	Device	Latency	Variability
File System	Short	100 ms	50ms
Git	Short	100 ms	100ms-1s
Web Search	Short	100 ms	50ms-2s
Database (SQLite)	Short	100-1000 ms	500ms
Web Search	Medium	1-5s	1-10s
AI Generation	Long	5-30s	10-60s

2.2 Challenges

The unique structure of multi-agent applications leads to two significant system-level challenges: space contention and time underutilization.

The heterogeneity in agent importance creates opportunities for space contention. As illustrated in Figure 2, an agent on the critical path, like *Node D*, can be stalled if a non-critical agent, like *Node C*, arrives earlier and occupies limited GPU memory. A simple memory allocation strategy may preempt the critical agent, forcing it into a costly context recomputation and delaying the entire workflow. Our analysis in Figure 2a confirms that these events are frequent in agentic workloads.

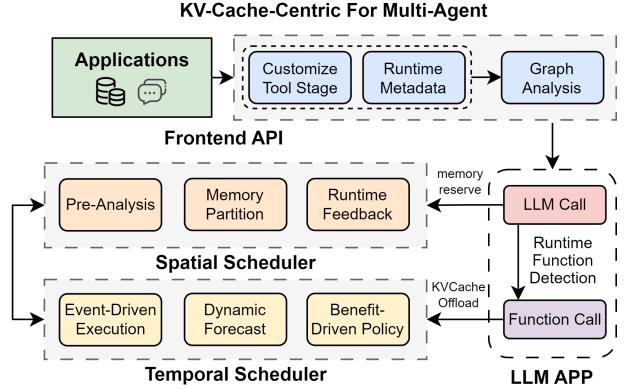
The reliance on external tools leads to severe time underutilization of GPU resources. Figure 3b illustrates the lifecycle of an agent’s KV Cache during a function call. After the first inference phase, the agent’s KV Cache is idle in GPU memory while it waits for the external tool to respond. This forces the serving system into a difficult trade-off: either retain the KV Cache, wasting valuable GPU memory, or evict it to serve other requests, forcing a costly recomputation when the agent resumes. Both choices result in inefficiency. This problem is substantial; as shown in Figure 3a, at peak moments, as much as 18.5% of the used KV Cache can be occupied by stalled agents.

2.3 Limitations of Existing Serving Systems

The challenges of space contention and time underutilization expose the limitations of two distinct categories of state-of-the-art serving systems: those that are agent-aware but compute-centric, and those that are KV-Cache-centric but agent-agnostic.

A class of recent work has made serving systems agent-aware by incorporating the application’s DAG into the scheduling logic. Systems like Parrot[8] and Autellix[10] use this graph structure to mitigate head-of-line blocking by prioritizing critical requests, while Teola[14] optimizes the execution pipeline for an individual agent’s interaction with external tools.

While these approaches improve high-level orchestration, their focus is fundamentally compute-centric. They optimize

**Figure 4.** Tokencake Overview.

the order and batching of requests but do not manage the underlying memory allocation. Consequently, they cannot prevent critical inversion, as a high-throughput but non-critical task can still occupy GPU memory and cause the eviction of a critical agent’s KV Cache. Furthermore, because they are not memory-centric, they do not address time underutilization, lacking the mechanisms to manage or repurpose the idle KV Cache of stalled agents.

Another line of work has focused on making serving more KV-Cache-centric, introducing advanced memory management and offloading policies. For instance, vLLM[7]’s Page-dAttention solves internal memory fragmentation, while systems like Mooncake[13] and CachedAttention[4] have implemented offloading for general workloads.

However, while these systems are memory-aware, their policies are fundamentally agent-agnostic. They treat all KV Cache with equal importance, lacking the context to differentiate a critical-path agent from a non-critical one, which leaves them vulnerable to critical inversion. Furthermore, their offloading policies are typically reactive—triggered by memory pressure or session inactivity—rather than proactive. They are not designed to leverage the predictable idle periods during function calls to mitigate underutilization.

3 Overview

Tokencake is designed to optimize multi-agent application performance by managing KV Cache resources across both time and space dimensions. Figure 4 shows the architecture of Tokencake, which comprises three primary components: a Frontend API, a Space Scheduler, and a Time Scheduler, which collaborate to orchestrate the lifecycle of KV Cache blocks in GPU memory.

The Frontend API (Section 3.1) translates user-defined application logic into an optimizable graph structure. This graph is then consumed by two specialized schedulers: The Space Scheduler (Section 5) implements a memory reservation policy to guarantee that critical agents receive their

```

1 rag_graph = Graph()
2 plan_node = LLMNode()
3 rag_node = RagNode(
4     stages=[Stage("query"), Stage("embed"),
5     Stage("generate")],
6     predict_time=1, config=rag_config)
7 revise_node = ReviseNode()
8 output_node = LLMNode()
9 rag_graph.add_edge(plan_node, rag_node)
10 rag_graph.add_edge(rag_node, revise_node)
11 rag_graph.add_edge(revise_node, output_node)

```

Figure 5. Example of defining a multi-agent RAG application with Tokencake’s API.

required KV Cache blocks, preventing memory-lack waiting for high-priority tasks. The Time Scheduler (Section 4) aims to maximize KV Cache utilization. It offloads an agent’s KV Cache to CPU memory during long-running external operations (e.g., a function call) and predictively uploads it back to the GPU for its next use, guided by an estimation of the external call’s execution time.

3.1 Frontend API

Tokencake provides a programming interface for users to represent multi-agent applications as a Directed Acyclic Graph (DAG). In this graph, nodes represent agents or computational units, while edges denote dependencies.

Figure 5 shows how to use this API to build a simple Retrieval-Augmented Generation (RAG) application. The user should first use the abstraction of node to define LLM functions. Line 3-5 shows the definition of a *RagNode*, which contains multiple internal stages. Then, by using the *add_edge* function, the user define the dataflow between multiple LLM agents (line 6). This process explicitly maps the application’s control flow and data dependencies into a graph structure that Tokencake can analyze.

Tokencake’s frontend API allows users to describe their multi-agent application as a Directed Acyclic Graph (DAG). In this representation, nodes correspond to agents or computational units, while edges define the data dependencies between them.

Tokencake provides a specialized *FuncNode* to represent the agents that make external function call. Unlike prior systems [8, 10, 14], Tokencake provides the ability to define fine-grained internal stages within a complex function call. In Figure 5, line 3-5, users can break down the *RagNode* into multiple sequential steps (“query”, “embed”, “generate”). This decomposition provides the scheduler with a more detailed, real-time view of the function’s progress, enabling more fine-grained scheduling decisions rather than treating the entire function as a monolithic, black-box operation.

Tokencake also provide the ability to supply critical performance metadata directly within the *FuncNode* definition. In Figure 5, line 5, the *predict_time* parameter allows a

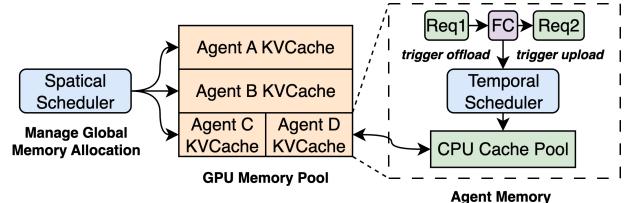


Figure 6. Coordination between the Space Scheduler and the Time Scheduler.

user to provide an estimated execution time for the function call, which is a crucial piece of information that will help the Time Scheduler make more accurate decisions about when to offload and prefetch an agent’s KV Cache. This ability to embed detailed application knowledge into the graph is essential for our co-optimization strategy.

3.2 Coordination between Space and Time Schedulers

Figure 6 shows the tight cooperation between the Space Scheduler and Time Scheduler.

While the Space Scheduler manages the global memory allocation between different agents, the Time Scheduler handles the memory lifecycle for a single agent’s request over time. Their coordination is most critical where their policies intersect.

The Space Scheduler handles the dynamic memory partition plan for agent types, using both application-level information and runtime feedback. When a request applies for KV Cache, the system first checks for memory availability according to the reservation policy established by the Space Scheduler for that request’s agent type.

After a request is scheduled to the GPU, the Time Scheduler begins to monitor its function call execution time. If the agent’s KV Cache is considered underutilized during this period, the Time Scheduler will offload it to CPU memory. This adjustment makes memory allocation more efficient for critical agents by freeing up GPU resources. The Space Scheduler then accounts for these offload and prefetch operations in its global management of the system’s memory.

4 The Time Scheduler

While the Space Scheduler manages memory contention *between* agents, the Time Scheduler addresses the underutilization caused by the function call’s execution time. A common pattern in agentic applications is a sequence of *LLM Inference1* \Rightarrow *Function Call* \Rightarrow *LLM Inference2*. In this sequence, the KV Cache generated during the first inference is a direct prefix for the second. However, during the function call, the KV Cache blocks that are not shared with other running requests remains idle in GPU memory, a state we call time underutilization. This presents a difficult trade-off:

Table 2. Comparison of KV Cache offloading and prefetching policies.

Category	Criteria	Tokencake	Mooncake	CachedAttn	LMCache
General	Function Call Aware	Yes	No	No	No
	Data Granularity	Block	Block	Layer	Block
Offload	Strategy	Proactive	Reactive	Reactive	Reactive
	Trigger	Function Call Start	Cache Pool Pressure	Session Inactive	Cache Eviction
	Decision Logic	Cost	LRU	Session	LRU
Prefetch	Strategy	Proactive	Proactive	Reactive	Reactive
	Trigger	Predicted FC Completion	SLO-based Schedule	Session Resumption	On-demand
	Decision Logic	Static and Dynamic	Static	Static	On-demand

evicting the cache frees up GPU memory but forces a costly recomputation later, while retaining it wastes resources that could be used by other active requests.

Offloading the underutilized KV Cache to CPU memory is a natural solution to this problem. Prior systems [4, 13, 18] have explored offloading, but their policies are designed for general server workloads, not for the predictable stalls unique to agentic workloads. As detailed in Table 2, these systems are not aware of function call, only reactive to the cache pressure or offloading a whole session. CachedAttention [4] triggers an offload when a conversation turn is complete and the session becomes inactive. Mooncake [13] and LMCache [18] make offload decisions based on general cache management policies, such as high memory pressure or evicting the least recently used items, rather than the specific state of an agent. In contrast, Tokencake’s policy is proactive and event-driven, a key advantage for agentic applications. It uniquely uses the Function Call event as an explicit trigger, allowing it to convert the predictable idle period into a planned scheduling window (Section 4.1).

Furthermore, when it’s time to bring the data back, reactive systems like CachedAttention and LMCache only start uploading the KV Cache when the request is ready to resume, which can introduce reactive overhead at a critical moment. Mooncake uses a profile-driven upload mechanism, loading the KV Cache that are most likely to be used next. But these systems are ignore the application-level information. Tokencake avoids this bottleneck by using a proactive upload mechanism. By forecasting the function call’s completion time, it begins loading the KV Cache back to the GPU before the agent needs it (Section 4.2). In this new, high-frequency offloading scenario, the data transfer overhead becomes a more critical performance factor, motivating the specialized optimizations we introduce to mitigate it (Section 4.3).

4.1 Event-Driven Offload and Predictive Upload

As illustrated in Figure 7, the Time Scheduler implements a complete lifecycle for managing the KV Cache, from static analysis to runtime execution and continuous learning.

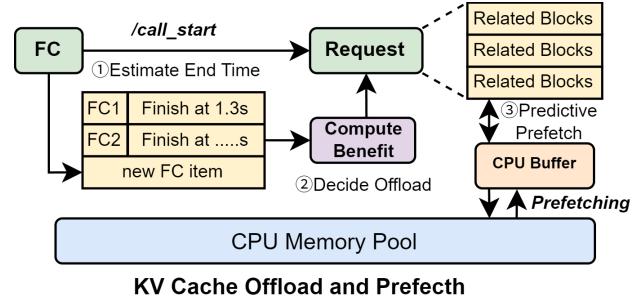


Figure 7. Lifecycle of the Time Scheduler’s offload and predictive upload mechanism.

Before the application runs, the scheduler performs an analysis on the static dependency graph to identify patterns like $LLM\ Inference1 \Rightarrow Function\ Call \Rightarrow LLM\ Inference2$. This step finds predictable periods of KV Cache underutilization and provides initial “cold-start” time predictions for function call the system has not encountered before, allowing the scheduler to make informed decisions based on the function’s type.

At runtime, the scheduler’s operations are driven by $call_start$ and $call_finish$ events from the inference engine. When a $call_start$ event is received, the scheduler consults the benefit-driven policy (Section 4.2) to decide whether to offload the agent’s KV Cache. If the decision is positive, it initiates an asynchronous transfer to CPU memory. As the function call nears its predicted completion time, the scheduler proactively uploads the KV Cache back to the GPU, aiming to hide the data transfer latency. This design is robust to prediction errors; an unexpected $call_finish$ event triggers an immediate upload to ensure correctness, and the observed execution time is fed back to the forecasting model to improve future predictions.

The scheduler’s predictive accuracy comes from its dynamic forecasting model. It begins with the cold-start estimate from the pre-runtime analysis and transitions to an

adaptive exponentially weighted moving average model after the first execution. When a developer provides a time estimate (t_{req}) in the graph definition, the model combines this hint with the system's historical data (t_{hist}):

$$t_{final} = \alpha \cdot t_{req} + (1 - \alpha) \cdot t_{hist} \quad (1)$$

4.2 Opportunistic Policy for Proactive Offloading

The decision to offload a stalled agent's KV Cache is governed by a predictive, opportunistic policy. An offload is only initiated if the anticipated benefit of freeing up GPU resources outweighs the cost of the data transfer. This core principle is expressed as: $Benefit_{scheduling} > Overhead_{transfer}$. The policy integrates several dynamic factors—such as the predicted function call duration, the size of the KV Cache, and the state of the waiting request queue—to evaluate this inequality in real-time.

Estimating KV Cache Transfer Overhead The primary overhead is the data transfer latency ($T_{transfer}$) for the movement of the KV Cache between the GPU and CPU memory. This cost is a direct function of the number of blocks (N_{blocks}) in the cache. We use an asynchronous CUDA implementation for memory transfers to avoid blocking the main scheduling loop. Our analysis in Section 7.4 shows that this transfer time is orders of magnitude smaller than the cost of recomputation. In Tokencake's scene, the number of KV Cache blocks is bounded to the model's context length, which is not very large. In this scenario, for a request with N_{blocks} blocks, the transfer time is generally linear with the number of blocks. We can estimate the total transfer time as:

$$Transfer = T_{offload}(N_{blocks}) + T_{upload}(N_{blocks})$$

Quantifying Scheduling Benefit. The benefit of offloading is not merely freeing memory, but rather the productive use of that memory and the associated compute resources during the stall period. Tokencake quantifies this benefit opportunistically: a benefit exists only if there is a waiting request that can be processed and completed within the time the stalled agent's resources are available.

The scheduler's decision logic is detailed in Algorithm 1. ① It first (line 7) calculates the available scheduling window (T_{window}) by subtracting the transfer overhead ($T_{transfer}$) from the predicted function call duration (T_{fc}). ② It then (line 8) converts this window into the number of tokens ($N_{capacity}$) that can be processed. ③ If a waiting request is found that fits within this capacity, the benefit is realized, and the offload is initiated (line 10). ④ If no such request exists, the KV Cache remains on the GPU, as there is no immediate benefit to offloading it.

This allows the new request to be admitted into the next batch, effectively hiding the function call latency of the first request and improving overall system throughput.

Algorithm 1 The Decision Logic of the Time Scheduler

```

1: procedure SHOULDOFFLOAD(req)
2:    $T_{transfer} \leftarrow \text{CalculateTransferTime}(\textit{req}.numblocks)$ 
3:    $T_{fc} \leftarrow \text{PredictFCDuration}(\textit{req}.FC)$ 
4:   if  $T_{fc} \leq T_{transfer}$  then
5:     return false                                 $\triangleright$  Stall is too short.
6:   end if
7:    $T_{window} \leftarrow T_{fc} - T_{transfer}$ 
8:    $N_{capacity} \leftarrow T_{window} \times v_{throughput}$      $\triangleright$  Computable
tokens.
9:    $\text{waiting\_req} \leftarrow \text{FindBestFitRequestInQueue}(N_{capacity})$ 
 $\triangleright$  Search for req where total  $\leq N_{capacity}$ .
10:  return  $\text{waiting\_req}$  is not null.  $\triangleright$  Decide offloading.
11: end procedure
```

4.3 Mitigating Offload and Predictive Upload Overhead

While Tokencake's proactive offload and upload mechanism is central to its performance, the operations themselves can introduce significant overhead if not managed carefully. This overhead originates from a key difference in memory management patterns for the GPU KV Cache block allocation and those offloaded to the CPU. On the GPU, KV Cache blocks are typically allocated incrementally as new tokens are generated by the scheduled running requests. The controlled batching of requests results in a steady, non-bursty pattern of memory operations. In contrast, blocks offloaded to the CPU follow a highly bursty lifecycle: an entire agent's KV Cache is allocated at the start of a function call and deallocated all at once upon its completion. This bursty pattern of CPU memory operations—allocating or freeing a large number of blocks simultaneously—can introduce significant latency that stalls the main scheduling loop.

To mitigate this latency, we introduce two targeted optimizations.

CPU Block Buffering. The lifecycle of offloaded KV Cache blocks induces high-frequency churn in CPU memory. When a function call begins, the unshared KV Cache blocks will be offloaded to the CPU, which leads to a large number of blocks must be allocated at once on the CPU. Conversely, when the call completes and the cache is uploaded, all associated CPU blocks are released simultaneously. Standard system memory allocators for this bursty allocation pattern are inefficient and can introduce significant latency due to system call overhead.

To address this, we implement a dedicated CPU block buffer. Instead of freeing blocks back to the operating system, Tokencake returns them to a lightweight, internal free list. Subsequent offload operations service their allocation requests from this buffer first, bypassing the costly system calls

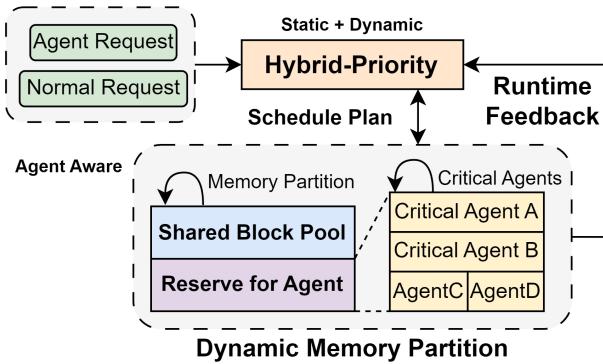


Figure 8. The Space Scheduler's dynamic memory partitioning feedback loop.

in the critical path. This design reduces the memory management latency for a large offload operation from nearly a second in worst-case scenarios to a consistent sub-millisecond level.

Gradual GPU Block Reservation. A core challenge in predictive uploading is guaranteeing GPU memory availability at the precise moment the data transfer must begin. As shown in Table 2, compared to reactive systems like Cache-dAttention and LMCache, which only fetch data when it's needed, Tokencake's proactive approach, which uses a predicted function call completion as its trigger, must solve the memory availability problem to be effective.

A naive "all-at-once" allocation of the required GPU blocks is risky. If the GPU memory is highly utilized or fragmented at that moment, the allocation can fail or stall, delaying the agent's execution and negating the benefits of predictive uploading. To make its proactive strategy reliable, Tokencake leverages its predictive model to perform gradual block reservation. Based on the function call's estimated completion time, the scheduler proactively begins reserving the required GPU blocks over several scheduling cycles before the upload is initiated. This approach amortizes the allocation process over time by making a series of smaller requests instead of one large one. This ensures that the destination blocks are ready when the predictive upload occurs, preventing allocation stalls and making the operation far more reliable.

5 The Space Scheduler

Agent workflows are often structured as dependency graphs, some agents are on the critical path, meaning their delay directly increases the application's total end-to-end latency. Prior works[8–10] notice the problem of the priority scheduling in LLM application scenario, but dismissing the KV Cache management.

However, a simple FCFS memory allocation policy is inefficient, if a non-critical agent occupies memory, it can stall a critical agent, creating a severe performance bottleneck.

The Space Scheduler is designed to solve this problem by managing memory with an awareness of each agent's importance. As shown in Figure 8, it uses a dynamic memory partitioning strategy that is guided by a hybrid priority metric.

5.1 Runtime Control with Dynamic Memory Partitioning

To protect critical agents, the scheduler implements a dynamic memory partitioning policy. It divides the GPU KV Cache memory into two regions: a globally shared pool available to all agents, and a reserved pool accessible only to the most critical agents. This policy ensures that even when the shared pool is heavily used, critical agents have guaranteed memory resources, preventing them from being stalled by less important tasks.

Critical Agent Selection. First, the scheduler periodically identifies which agents are currently "critical." It does this by calculating a combined hybrid priority score for every agent type and designating the top fraction (a configurable *critical_ratio*) as critical. This ensures the set of protected agents adapts to changing application needs.

Dynamic Memory Partitioning. Once the critical agents are identified, the scheduler updates the memory reservations according to the two-phase process in Algorithm 2.

First, in Phase 1 (lines 5–11), the algorithm adjusts the *total_size* of the reserved memory pool. It does this based on system-wide memory pressure, calculated from the current GPU block usage ratio. If memory usage is high, the *total_reserve_ratio* is increased to provide more protection for critical tasks. If usage is low, the ratio is decreased to avoid wasting memory that could be used by the shared pool.

Second, in Phase 2 (lines 13–19), the algorithm partitions this total reserved pool (*R_total*) among the individual critical agents. An agent's share is a weighted average of two factors: the agent's historical memory usage (line 15) and its relative priority score (line 16). This balanced approach ensures that agents that are both important and typically memory-intensive receive a proportionally larger reservation.

5.2 Hybrid Priority Metric for Agent-Awareness

Tokencake moves beyond simple FIFO scheduling, using a hybrid priority plan, which considers both an application's static structure and its dynamic runtime state.

The static priority measures an agent's structural importance within the application's Directed Acyclic Graph (DAG). The formula is:

$$\text{priority}_{\text{static}} = w_{\text{static}} \times \text{node}_{\text{depth}} \times \text{node}_{\text{out_degree}}$$

Algorithm 2 Memory Reservation Update

```

1: procedure UPDATERMEMORYRESERVATIONS
2:   usage ← GetLastGpuBlockUsage()
3:   tot_blk ← GetTotalGPUBlocks()
4:   ▷ Phase 1: Adjust the reserved memory pool size
5:   ratio ← usage / tot_blk
6:   if ratio ≥ gpu_usage_high then
7:     total_reserve_ratio += adjustment_step
8:   else if ratio ≤ gpu_usage_low then
9:     total_reserve_ratio -= adjustment_step
10:  end if
11:  Rtotal ← tot_blk × total_reserve_ratio
12:  ▷ Phase 2: Partition the pool among critical agents
13:  Stotal ←  $\sum_{a \in \text{critical\_agents}} \text{GetAgentScore}(a)$ 
14:  for all agent_type in critical_agents do
15:    mem_ratio ← GetUsage(agent_type) / tot_blk
16:    priority_ratio ← GetScore(agent_type) / Stotal
17:    final_ratio ← (mem_ratio + priority_ratio) / 2
18:    reserve_num[agent_type] ← final_ratio × Rtotal
19:  end for
20: end procedure

```

The weight w_{static} is a configurable parameter that normalizes the static priority score relative to the dynamic priority score, balancing structural importance with runtime urgency.

The dynamic priority adapts to changing runtime conditions to balance system throughput and fairness. It is calculated based on how long a request has been waiting ($time_{wait}$) and the number of tokens to be processed. The linear term for waiting time ensures older requests are eventually scheduled, while the logarithmic term for token count gives a modest preference to shorter requests. The formula is:

$$\text{priority}_{dynamic} = time_{wait} \times \log\left(\frac{tokens_{req}}{time_{wait}}\right)$$

6 Implementation

Tokencake is a KV-Cache-centric serving system for multi-agent applications, composed of a front-end and an execution engine. It is implemented in approximately 9k lines of Python code, uses Triton for custom kernels, and reuses some components from vLLM.

Frontend. Tokencake's frontend extends OpenAI's Chat Completion to provide a stateful interface that appears stateless to developers. The frontend allows developers to define multi-agent workflows as a Directed Acyclic Graph (DAG). In this graph, nodes represent agents or specific computational units, while edges define the data dependencies and control flow between them. Each node can be configured with specific metadata, such as the LLM model to use or maximum token limits, providing fine-grained control over each

Table 3. FuncNode types available in the Tokencake API.

Node	Description
FileReadNode	Read the contents of a specified file.
FileWriteNode	Write content to a specified file.
SearchNode	Perform a search for a given query.
FileQueryNode	Query files under a specified path.
DataAnalysisNode	Multi-stage analysis of large datasets.
UserConfirmNode	Request user confirmation.
ExternalTestNode	Use external test tools.

step of the workflow. As shown in Table 3, Tokencake provides many common pre-built nodes for common functions to streamline development.

Developers can construct the graph using a set of predefined and customizable node types.

Execution Engine. The engine tracks the lifecycle of function calls through two API endpoints to inform the scheduler. When an application begins a function call, it notifies the engine by sending a request to `call_start`. This request includes an initial estimate of the call's execution time, which helps the scheduler make an immediate decision. Once the function call completes, the application sends a request to `call_finish`. This second notification signals that the suspended request is ready to run again and provides the actual execution time, which is used to refine the engine's prediction model for future calls of the same type.

Asynchronous KV Cache Management. A key challenge is performing the offload and prefetch operations without blocking the main scheduling loop. Synchronous data transfers would introduce significant latency, defeating the purpose of the optimization. Therefore, all KV Cache migration is implemented asynchronously. We use custom CUDA kernels for the physical data transfers between GPU and host memory.

The management logic is integrated with the engine's scheduling loop. At the start of each scheduling cycle, before a new batch is formed, the engine initiates two sets of asynchronous memory transfers. First, it identifies requests whose function calls are predicted to finish soon and begins proactively moving their KV Cache blocks from CPU memory back to the GPU. Second, it processes any new offload decisions by moving the corresponding KV Cache blocks from the GPU to the CPU. The source GPU blocks are marked as pending free and are only returned to the memory pool after the transfer is complete. Each memory block maintains a state flag to track its current location, whether in GPU or host memory.

7 Evaluation

In this section, we present a comprehensive evaluation of Tokencake. We first assess its effectiveness in minimizing

the end-to-end latency of multi-agent applications under various loads. We then examine each component of Tokencake to understand its specific contribution to the overall performance improvement.

7.1 Experimental Setup

Model and Server Configurations. We evaluate Tokencake using two versions of the Qwen2.5 model. The 14B parameter model runs on a server with an NVIDIA A100 GPU (80GB), while a larger 32B model runs on a server with an NVIDIA H200 GPU (140GB). For Tokencake's offloading feature, we set aside 100GB of CPU memory as swap space to store the evicted KV Cache blocks.

Benchmark Applications. To ensure our evaluation reflects realistic scenarios, we implement two representative multi-agent applications: "Code-Writer" and "Deep Research", shown in Figure 1. The Code-Writer application (Figure 1a) is characterized by a large number of specialized agents (e.g., programmer, reviewer, tester) and frequent tool usage (e.g., file I/O), creating significant memory pressure and testing the system's capacity for managing numerous concurrent KV Cache states. The Deep Research application (Figure 1b) models a research process involving steps like planning, searching, and summarizing. While it involves fewer agents than Code-Writer, it features a more intricate control flow with complex dependencies, challenging the scheduler's ability to optimize the critical path and manage inter-agent stalls effectively.

Workload. User requests for the benchmarks are synthesized from the ShareGPT[2] and AgentCode[1] datasets, which contain real-world conversational data. To simulate a dynamic user environment, we generate request arrivals using a Poisson distribution, varying the rate of applications per second to evaluate system performance under different load conditions.

For tool-using agents, because the model we use cannot steadily generate tool using output, the external function calls are simulated to ensure a controlled and repeatable evaluation of the serving system's performance. The latencies for these function calls are also modeled with a Poisson distribution, which aligns with the behavior documented in the Model Context Protocol (MCP). This simulation methodology allows us to isolate and accurately measure the performance of our scheduling and memory management framework, independent of the non-deterministic nature of model-generated outputs.

Baseline. We compare Tokencake against two state-of-the-art LLM serving frameworks: vLLM and LightLLM. The vLLM baseline represents the typical behavior of current serving systems, where the KV Cache of a request blocked by a function call remains resident in GPU memory, leading to inefficient resource use during the stall period. LightLLM

features a lightweight architecture with a fine-grained, token-wise memory manager. However, while its scheduler is optimized for high token throughput, it is not designed to handle the long, unpredictable idle times inherent to agentic workloads that use external tools.

Neither of these baseline systems includes the proactive offloading or predictive uploading mechanisms that are central to our design. This comparison allows us to clearly isolate and highlight the performance gains achieved by Tokencake's KV-Cache-centric approach, which is tailored specifically for the challenges of multi-agent applications.

Metrics. We measure End-to-End Latency, the total time from a user request's submission to the reception of the final response, reporting both average latency. To assess memory efficiency, we track GPU KV Cache Utilization, the percentage of GPU memory blocks allocated to the KV Cache over time. Finally, to quantify blocking and contention on the critical path, we use the Abnormal Agent Count, defined as the number of agent instances whose execution time exceeds 1.5 times the average for their type.

7.2 Performance Results

End-to-End Latency. To evaluate Tokencake's ability to handle varying workloads, we measured the average end-to-end latency under different request rates, simulated using a Poisson distribution. As shown in Figure 9, the performance benefits of Tokencake become increasingly pronounced as the system load increases.

At lower request rates (e.g., 0.05 QPS), Tokencake and the vLLM baseline exhibit comparable performance, as memory contention is not yet a significant issue. However, LightLLM shows worse performance even at this low load. LightLLM features a fine-grained, token-wise memory manager optimized for high token throughput. While effective for continuous generation, its architecture is not designed to handle the long and unpredictable idle times that occur when agents use external tools. This architectural mismatch leads to higher initial latency. As the QPS rises and function calls become more frequent, the latency for both baseline systems grows much more rapidly. The core issue is that both vLLM and LightLLM keep the KV Cache of stalled agents resident in the GPU, quickly leading to memory saturation. This contention forces their schedulers to use smaller, less efficient batch sizes and delays the processing of new and existing requests.

In contrast, Tokencake's latency scales much more gracefully with the increasing load. By intelligently offloading the KV Cache of stalled agents, Tokencake frees up valuable GPU memory, allowing it to maintain larger and more efficient batch sizes for active requests. This proactive memory management prevents the system from becoming a bottleneck. For instance, at a high load of 1.0 QPS, Tokencake reduces the average end-to-end latency by over 47.06% compared to the vLLM baseline. This result clearly demonstrates Tokencake's

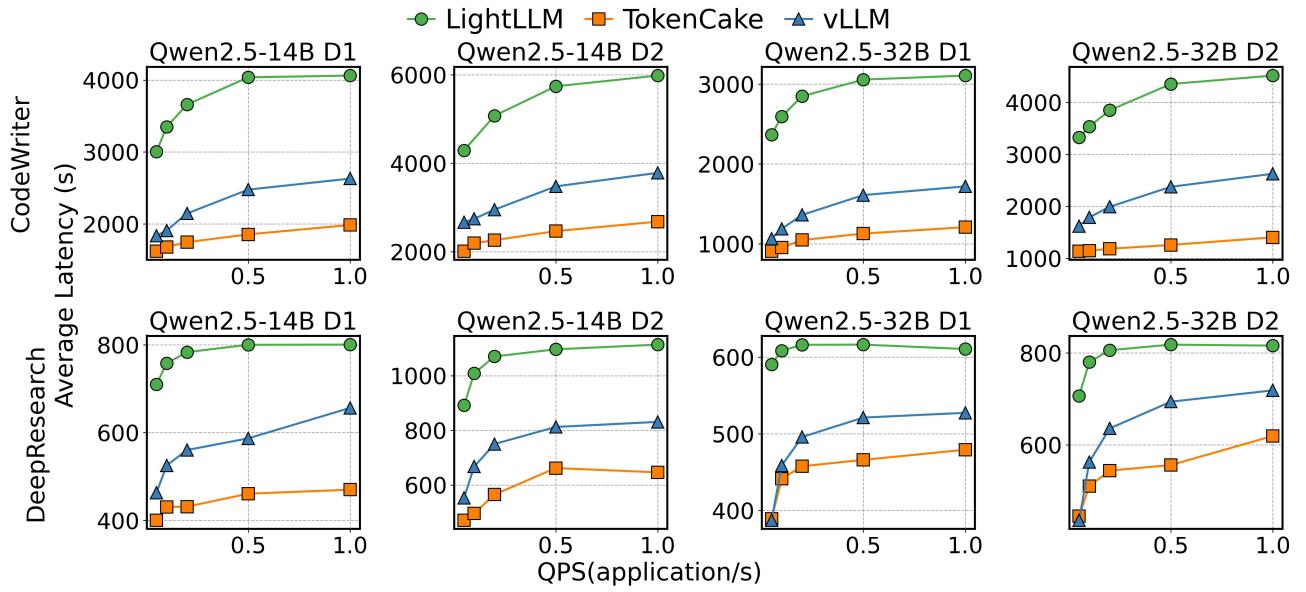


Figure 9. End-to-end application latency comparison of Tokencake, vLLM, and LightLLM. Each chart plots average latency against queries-per-second (QPS) for the specified application, model, and dataset.

superior ability to maintain high performance and stability under the demanding conditions of multi-agent workloads, a scenario where other specialized systems like LightLLM falter.

GPU Utilization and Memory Management. The latency improvements in Tokencake are a direct result of its more efficient memory management, which leads to higher effective GPU utilization. As illustrated in Figure 10, Tokencake consistently maintains a higher average GPU KV Cache usage—hovering around 86 – 87% across all load levels—which is up to 16.9% higher than vLLM.

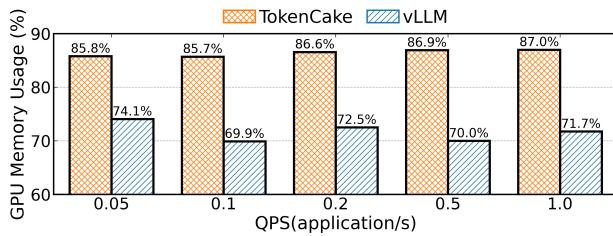


Figure 10. GPU KV Cache utilization under varying load.

Tokencake's proactive offloading policy ensures that the GPU memory is predominantly occupied by the KV Cache of active, computation-ready requests. By intelligently moving the caches of agents stalled on function calls to CPU memory, Tokencake frees up valuable GPU resources that can be immediately repurposed. This allows the system to

sustain larger, more computationally efficient batch sizes and process more requests concurrently.

In contrast, the baseline system's lower utilization reflects a critical inefficiency. In vLLM, the memory quickly becomes fragmented with a mix of active and idle KV Cache from stalled agents. While its memory is also occupied, a significant portion is held by these underutilized caches, which are not contributing to active computation but are instead blocking new requests from being scheduled. This memory contention forces the baseline scheduler to use smaller batch sizes, leading to lower throughput and memory utilization.

This improved memory management translates directly to higher computational efficiency and generation throughput. By resolving the memory bottleneck caused by idle caches, Tokencake enables the GPU's computational resources to be used more effectively, ultimately processing more tokens per second across the entire workload.

7.3 Agent Analysis: Optimizing the Critical Path

Agent Latency Comparison. Beyond improving overall latency, Tokencake enhances the performance of individual agents by reducing system-wide resource contention. As shown in Figure 11, every agent type runs faster on Tokencake. However, the key to minimizing end-to-end application time is optimizing the critical path. Baseline systems are prone to priority inversion, where a low-priority agent stalls a critical one, creating a significant bottleneck. Tokencake's agent-aware Space Scheduler prevents this by using a dynamic memory reservation policy to guarantee resources

for critical-path agents. This ensures the most important tasks proceed without delay, balancing the entire workflow for a shorter total execution time. The effectiveness of this strategy is confirmed by the sharp reduction in "abnormal agents" (agents with unusually long execution times), as seen in Figure 12.

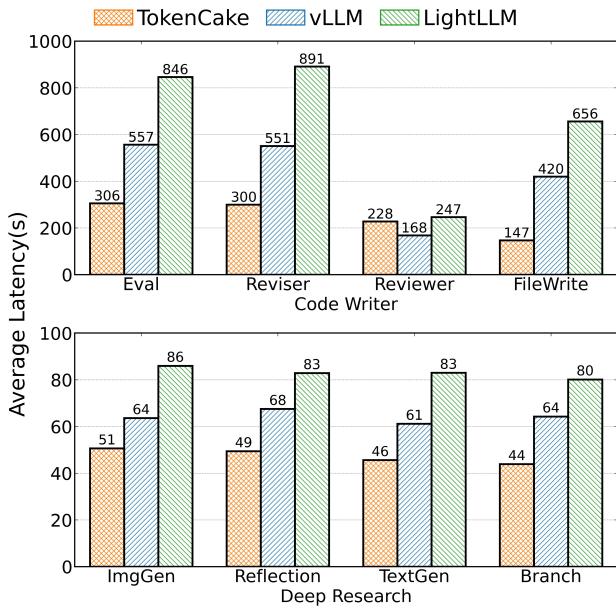


Figure 11. Average latency by agent type

Critical Path Optimization. To quantify how Tokencake optimizes the application workflow, we analyze the number of "abnormal agents," which are defined as agents whose execution time is more than 1.5 times the average for their type. A high count of these latency outliers suggests frequent blocking and resource contention, a problem that is particularly damaging when it delays agents on the critical path. As shown in Figure 12, Tokencake significantly reduces the number of abnormal agents compared to the baseline systems. This result demonstrates that Tokencake's agent-aware scheduling and memory management effectively mitigate the worst-case delays caused by contention. By ensuring resources are available for critical tasks, Tokencake creates a smoother and more stable execution flow, which is essential for achieving reliable performance in complex multi-agent applications.

7.4 Analysis of the Offloading Tradeoff

Tokencake is based on the idea that moving a stalled agent's KV Cache to CPU memory is much faster than recomputing it later. To test this, we measured the time cost of both actions. Figure 13 compares the time for transferring the KV Cache (offload from GPU to CPU and upload back) with the time for recomputation. The data clearly shows that data

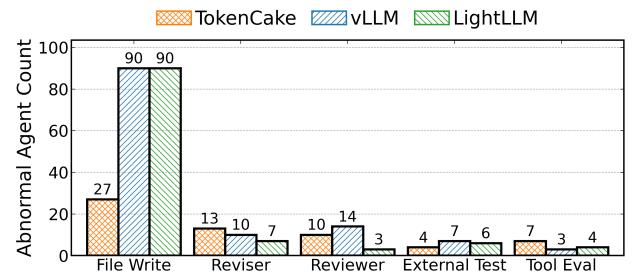


Figure 12. Reduction in the count of abnormal agents. An agent is considered abnormal if its execution time exceeds 1.5x the average for its type.

transfer is orders of magnitude faster than recomputation. For example, transferring 4096 blocks takes about 60 ms, while recomputing them takes nearly 9,000 ms. This large time difference confirms that our approach is efficient. The overhead of moving the KV Cache is very small compared to the high cost of recomputation, even for long function call stalls. This allows Tokencake to free up GPU memory for active tasks without a major penalty when the stalled agent resumes, creating a good balance between memory availability and resumption speed.

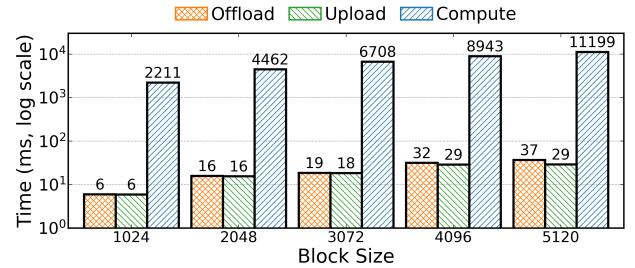


Figure 13. Time tradeoff between KV Cache reuse and recomputation.

Offload Overhead Mitigation The performance of Tokencake's time scheduling hinges on the efficiency of its offload and upload operations. The high frequency of these transfers means that any associated overhead could negate the benefits of freeing up GPU memory. We designed two key optimizations to address this: CPU Block Buffering and Gradual GPU Block Reservation (Section 4.3). To quantify their impact, we conducted a micro-benchmark comparing Tokencake with these optimizations against a baseline version without them.

As shown in Figure 14, the results demonstrate the critical importance of these mitigation techniques. The baseline version incurs prohibitively high latency, scaling from 4,366 ms for 1,024 blocks to an overwhelming 15,163 ms for 5,120 blocks, which stems from the inefficient handling of bursty

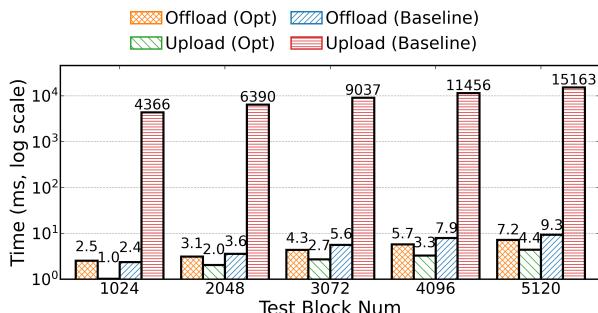


Figure 14. Overhead Mitigation for KV Cache Offload and Upload Operations.

memory allocation requests. In contrast, the optimized version of Tokencake reduces this overhead by several orders of magnitude, with both offload and upload latencies remaining in the single-digit milliseconds. For 5,120 blocks, the upload time is reduced from 15,163 ms to just 4.4 ms. This dramatic improvement confirms that our CPU block buffering and gradual GPU reservation strategies are essential, transforming the data transfer from a major bottleneck into a lightweight operation and making Tokencake’s proactive, high-frequency offloading strategy viable and highly effective.

8 Related Works

LLM Application-Aware Scheduling. Several systems optimize multi-agent application performance by making the scheduler aware of the application’s structure. For instance, Parrot [8] and Autellix [10] treat the application as a graph to prioritize requests and mitigate head-of-line blocking. Teola [14] optimizes the execution pipeline for an individual agent’s interaction with external tools like RAG. While these approaches improve high-level orchestration, they are largely ignore the memory management problem. They operate at the request level and do not manage the underlying KV Cache, leaving systems vulnerable to performance issues like priority inversion caused by space memory contention. In contrast, Tokencake’s Space Scheduler directly manages memory allocation with a dynamic partitioning policy to protect critical-path agents, resolving contention at its source.

KV Cache Optimization. KV-Cache-centric has become a new paradigm for more efficient KV Cache management. Mooncake [13], a disaggregated serving architecture, treats KV Cache as first-citizen across different stage of inference. Recent systems have introduced offloading mechanisms to optimize memory usage. CachedAttention [4] used offloading to reduce the TTFT time. While effective for general workloads, these offload policies are typically reactive and not designed for the frequent, predictable stalls inherent

to agentic applications. Tokencake addresses this specific challenge by introducing a proactive policy that leverages application-level events—namely, function calls—to anticipate idle periods and manage the KV Cache lifecycle accordingly.

9 Discussion and Future Work

Our evaluation demonstrates the benefits of Tokencake’s design, though our work also has limitations that point to several directions for future work.

One limitation is that Tokencake’s scheduling policy relies on a simple model to predict tool execution times. The design is robust to prediction inaccuracies. An early call_finish event triggers an immediate prefetch to ensure correctness, and the observed execution time is used to improve future predictions. Furthermore, our opportunistic policy only initiates an offload when there is a clear scheduling window and a waiting request that can be completed within it, guaranteeing a performance benefit even with imperfect time estimates. This mechanism opens a promising direction for future work in the co-design of more sophisticated scheduling policies and prediction models. For instance, a scheduler could incorporate richer predictive features, such as function call arguments, to better balance system throughput with fairness.

A second limitation is the single-GPU scope of our current evaluation. We believe that the core principles of agent-aware, dynamic KV Cache management are directly applicable to larger, distributed environments. Extending Tokencake to a multi-GPU setup is a natural next step. The space and time scheduling could be adapted to manage a tiered memory hierarchy, using a neighboring GPU’s memory over high-speed interconnects like NVLink as a faster offload target than CPU RAM.

10 Conclusion

This paper introduced Tokencake, an LLM serving framework designed to solve performance problems in multi-agent applications. Long-running function calls often cause an agent’s KV Cache to idly occupy valuable GPU memory, leading to underutilization and high latency.

Tokencake tackles this by dynamically offloading the idle KV Cache to CPU memory and using predictive uploading to hide data transfer latency when the agent resumes. Our evaluation on a realistic multi-agent benchmark shows that Tokencake significantly reduces end-to-end latency by up to 47.06% compared to a standard vLLM baseline. These results show the benefits of making the serving system aware of the application’s context, enabling more efficient and responsive agentic applications.

References

- [1] AlignmentLab AI. 2024. AgentCode: A dataset for code generated by LLM agents. Hugging Face Datasets. <https://huggingface.co/datasets/AlignmentLab-AI/agentcode> Accessed: 2025-09-01.
- [2] anon8231489123. 2023. ShareGPT Vicuna Unfiltered. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered Hugging Face dataset.
- [3] Anthropic, Inc. 2025. *Model Context Protocol Specification*. <https://spec.modelcontextprotocol.io/specification/2025-08-20/> Accessed: 2025-08-20.
- [4] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-efficient large language model serving for multi-turn conversations with CachedAttention. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) (USENIX ATC'24). USENIX Association, USA, Article 7, 16 pages.
- [5] Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, Liuyi Yao, Hongyi Peng, Zeyu Zhang, Lin Zhu, Chen Cheng, Hongzhu Shi, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. AgentScope: A Flexible yet Robust Multi-Agent Platform. arXiv:2402.14034 [cs.MA] <https://arxiv.org/abs/2402.14034>
- [6] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. arXiv:2308.00352 [cs.AI] <https://arxiv.org/abs/2308.00352>
- [7] 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* (Koblenz, Germany) (SOSP '23). Association for Computing Machinery, New York, NY, USA, 611–626. <https://doi.org/10.1145/3600006.3613165>
- [8] 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>
- [9] Yifei Liu, Zuo Gan, Zhenghao Gan, Weiye Wang, Chen Chen, Yizhou Shan, Xusheng Chen, Zhenhua Han, Yifei Zhu, Shixuan Sun, and Minyi Guo. 2025. Efficient Serving of LLM Applications with Probabilistic Demand Modeling. arXiv:2506.14851 [cs.DC] <https://arxiv.org/abs/2506.14851>
- [10] 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>
- [11] Microsoft. 2023. Microsoft 365 Copilot. Web page. <https://www.microsoft.com/en-us/microsoft-365/enterprise/microsoft-365-copilot>
- [12] Jinghua Piao, Yuwei Yan, Jun Zhang, Nian Li, Junbo Yan, Xiaochong Lan, Zhihong Lu, Zhiheng Zheng, Jing Yi Wang, Di Zhou, Chen Gao, Fengli Xu, Fang Zhang, Ke Rong, Jun Su, and Yong Li. 2025. AgentSociety: Large-Scale Simulation of LLM-Driven Generative Agents Advances Understanding of Human Behaviors and Society. arXiv:2502.08691 [cs.SI] <https://arxiv.org/abs/2502.08691>
- [13] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. arXiv:2407.00079 [cs.DC] <https://arxiv.org/abs/2407.00079>
- [14] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2025. Teola: Towards End-to-End Optimization of LLM-based Applications. arXiv:2407.00326 [cs.DC] <https://arxiv.org/abs/2407.00326>
- [15] Google Gemini Team. 2025. Gemini Fullstack LangGraph Quickstart. <https://github.com/google-gemini/gemini-fullstack-langgraph-quickstart>. Accessed: 2025-09-23.
- [16] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs.AI] <https://arxiv.org/abs/2308.08155>
- [17] Yijia Xiao, Edward Sun, Di Luo, and Wei Wang. 2025. TradingAgents: Multi-Agents LLM Financial Trading Framework. arXiv:2412.20138 [q-fin.TR] <https://arxiv.org/abs/2412.20138>
- [18] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*. 94–109. <https://doi.org/10.1145/3689031.3696098>
- [19] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. arXiv:2401.07339 [cs.SE] <https://arxiv.org/abs/2401.07339>