RTP-LLM: High-Performance Alibaba LLM Inference Engine

[†]Boyu Tan, [§]Jiarui Guo, [§]Zongwei Lv, [‡]Haobo Sun, [§]Tong Yang, [†]Kan Liu, [†]Xinfei Shi, [†]Zetao Hu, [†]Yaxin Yu, [†]Chi Zhang, [†]Jianning Zhang, [†]Xi Yang, [†]Wei Zhang, [†]Bo Cai, [†]Silu Zhou, [†]Xiyu Wang, [†]Na He, [†]Yinghao Yu, [†]Wending Bao, [†]Guiyang Huang, [†]Yuxing Yuan, [†]Juncheng Yin, [†]Nan Wang, [†]Lin Yang, [†]Zechao Zhang, [‡]Lu Chen, [†]Guoding Li, [†]Tao Lan, [†]Lin Qu

[†]Alibaba Group [§]Peking University [‡]Zhejiang University

[†]{tanboyu.tby, xinfei.sxf, huzetao.hzt, xieshui.yyx, yujing.zc, zhangjianning.zjn, wangyin.yx, zw193905, qisa.cb, silu.zsl, xiyu.wxy, yinghao.yyh, baowending.bwd, cangfei.hgy, yuanyuxing.yyx, yinjuncheng.yjc, kenan.wn,ziyu.yl, zhangzechao.zzc, guoding.lgd, tao.lant, xide.ql}@alibaba-inc.com [†] luoli.hn@taobao.com [§] {ntguojiarui, lvzongwei, yangtong}@pku.edu.cn [‡]{haobosun, luchen}@zju.edu.cn

arXiv:2605.29639v1 [cs.OS] 28 May 2026

Abstract

Large Language Models (LLMs) have revolutionized AI applications, but deploying them at scale presents significant challenges. We present RTP-LLM, a high-performance inference engine for industrial-scale LLM deployment, successfully deployed across Alibaba Group serving over 100 million users. RTP-LLM addresses fundamental bottlenecks through integrated design. It optimizes model loading via file-order-driven I/O and parallel I/O-communication overlapping. The Prefill-Decode Disaggregation architecture decouples compute-intensive prefill from memory-bound decode phases, combined with hierarchical multi-tiered KV cache management enabling efficient cache reuse. In addition, RTP-LLM incorporates modular speculative decoding supporting multiple algorithms, adaptive KV cache quantization, and decoupled multimodal processing, with support for multi-level parallelism.

Comprehensive evaluations across diverse model architectures (8B-235B parameters) have been conducted, where both controlled benchmarks and real production workloads are used. The results demonstrate RTP-LLM's superior performance against vLLM and SGLang: 4.7x-6.3x model loading speedup, 35-37% TTFT P95 latency reduction with 215% cache reuse improvement in production traffic scheduling, 1.12x-2.48x and 1.86x-2.52x throughput improvements in speculative decoding and multimodal inference, respectively, and 35-40% batch latency reduction with 1.9x-3.0x TTFT improvement in quantized inference. RTP-LLM's production-proven architecture and open-source availability make it a comprehensive solution for industrial LLM deployment.

1 Introduction

The rapid advancement of Large Language Models (LLMs) has catalyzed a paradigm shift across industries, transforming applications ranging from conversational AI and code generation to enterprise automation [1, 17, 20]. Modern models have reached a scale of hundreds of billions of parameters [7, 35], delivering unprecedented capabilities but also imposing severe computational and memory demands that traditional inference systems were never designed to handle. This chasm between model scale and deployability has emerged as a critical bottleneck, requiring fundamental rethinking of system architecture rather than incremental optimization.

The challenge is fundamentally rooted in the autoregressive nature of LLM inference, where each generated token depends on all preceding tokens through sequential attention computations [50]. Unlike traditional machine learning workloads that benefit from embarrassingly parallel batch processing, autoregressive generation creates an intrinsic sequential bottleneck that fundamentally limits GPU utilization and throughput [3, 56]. Compounding this issue, the Key-Value (KV) cache—a critical data structure for storing intermediate attention states—grows dynamically during generation and can dominate memory consumption, particularly as context lengths exceed 128K tokens [15, 61, 63]. Early inference frameworks treat these constraints as immutable, resulting in suboptimal resource utilization and prohibitive latency for production deployments.

Recent years have witnessed significant innovations targeting specific facets of this problem. Systems like **vLLM** introduced PagedAttention, revolutionizing memory management by treating the KV cache as a paged virtual memory system [30]. **TensorRT-LLM** delivered kernel-level optimizations tailored to NVIDIA hardware [40], while **FlashAttention** restructured attention computations to reduce memory bandwidth pressure [12, 13]. Despite these advances, a critical gap persists: most existing solutions optimize isolated components while neglecting the systemic interactions required for production-scale deployment. They typically focus on single-node performance, lack support for heterogeneous hardware, and fail to address the full spectrum of challenges—from dynamic workload scheduling to rapid model iteration—that enterprises face in practice [27, 28, 32].

Production LLM deployments face four fundamental challenges that existing systems inadequately address:

Challenge I: Underutilized GPUs under dynamic sequential workloads. Request patterns exhibit extreme variability in input length (from short queries to 128K+ contexts), output length, and computational intensity [2, 53]. The memory-bound nature of autoregressive attention leaves compute units idle, while static batching strategies cannot adapt to request dynamism, resulting in unpredictable latency and suboptimal throughput.

Challenge II: Memory exhaustion from unconstrained KV cache growth. The KV cache grows linearly with sequence length and batch size, quickly becoming the dominant memory consumer [31, 36]. Traditional memory allocators struggle with fragmentation and inefficient sharing across requests with diverse context lengths. As modern models support ever-longer contexts, memory management has become a hard capacity constraint that limits concurrency and scalability.

Lu Chen is the corresponding author.

Challenge III: System rigidity in the face of architectural heterogeneity. Modern model architectures introduce significant complexity that existing systems struggle to handle efficiently. Large-scale Mixture-of-Experts (MoE) models exceeding 600B parameters require efficient expert routing and rapid weight loading [35], while multi-modal models combine vision encoders and language models that have fundamentally different computational characteristics and execution patterns, requiring coordinated execution across heterogeneous computation components.

Challenge IV: Operational fragility impeding rapid iteration. Enterprise deployments demand minute-level loading of 600B+ parameter models to enable continuous updates across business units [42]. Existing systems require hours for weight loading and lack production-grade fault tolerance, rolling update mechanisms, and per-request performance isolation needed to meet stringent latency SLOs under fluctuating load.

To address these challenges, we present **RTP-LLM**, a holistic inference system developed by Alibaba's Foundation Model Inference Team and battle-tested across production deployments serving Taobao, Tmall, and Cainiao.

To enable rapid model iteration (addressing Challenge IV), RTP-LLM implements **Optimized Model Loading** (Section 4) through file-order-driven I/O, shared memory reuse, and parallel I/O-communication overlap, achieving 1.4x-6.3x faster loading times compared to vLLM and SGLang, enabling minute-level deployment of 600B+ parameter models. RTP-LLM incorporates enterprise-grade operational features including fault tolerance, rolling updates, and per-request performance isolation to ensure predictable latency under stringent SLOs.

To maximize GPU utilization and memory efficiency (addressing Challenge I and II), RTP-LLM implements **Prefill-Decode Disaggregation** [41, 43, 66] (Section 5) that physically decouples compute-intensive prefill from memory-bound decode phases, enabling independent scaling and optimal resource allocation. **Dynamic Traffic Scheduling** with intelligent load balancing continuously reprioritizes requests based on queue state, KV cache footprint, and latency targets, maximizing GPU utilization through dynamic batching [22, 59]. **Hierarchical Multi-Tiered KV Cache Management** (Section 5) spans GPU memory, local CPU memory, remote CPU memory via RDMA, and distributed storage, with unified hash-based prefix matching enabling efficient cache reuse. **Prefix Caching** enables fine-grained reuse of KV cache pages across requests sharing common prefixes (e.g., system prompts, RAG passages), significantly reducing computational overhead [26, 64, 65]. Production evaluations demonstrate 35-37% TTFT P95 latency reduction and 215% cache reuse improvement, enabling 75% reduction in prefill machine count.

To break the sequential generation bottleneck (further addressing Challenge I), we employ **Multi-Token Speculative Decoding** (Section 6) supporting multiple algorithms (Medusa, Eagle, Prompt Lookup) that achieve 1.12x-2.48x throughput improvement by predicting and verifying multiple future tokens in parallel while preserving output quality [46, 54].

To support diverse model architectures (addressing Challenges II and III), RTP-LLM implements **Adaptive KV Cache Quantization** (Section 7) that reduces memory footprint and improves the performance. **Multi-Level Parallelism** (Section 7) integrates Tensor, Data, Pipeline, and Expert Parallelism to support diverse model architectures from dense models to MoE models exceeding 600B parameters. For multimodal models, **Decoupled ViT-LLM Processing** (Section 7) separates vision encoding from language generation, achieving 1.86x-2.52x throughput improvement and 2.12x-2.36x TTFT reduction.

This paper makes the following contributions:

- We present the design and implementation of RTP-LLM, a comprehensive inference system that integrates memory management, scheduling, and hardware acceleration into a cohesive, production-ready platform. RTP-LLM addresses the full inference stack with battle-tested reliability across Alibaba's ecosystem, serving over 100 million users in production environments.
- We describe system design and engineering practices including hierarchical load balancing for disaggregated serving, unified multi-modal request orchestration, and adaptive resource allocation schemes that dynamically adjust GPU memory fractions between prefill and decode engines based on live traffic analysis.
- We provide extensive performance evaluations across diverse model architectures (dense, MoE, multi-modal) using both controlled benchmarks and **real production workloads**, offering actionable insights into the effectiveness and interaction of different optimization techniques in real-world deployment.
- We release RTP-LLM as open-source software [4], providing an extensible foundation for research and development in LLM inference. The system has already attracted active community adoption, fostering further innovation in the field.

2 Background

As model sizes (i.e., #parameters) of LLMs have grown from millions [10, 29, 44] to hundreds of billions [7, 35], the computational and memory requirements for real-time inference have escalated exponentially, making it challenging to deploy LLMs in production environments. The current landscape of LLM inference systems is characterized by a diverse ecosystem of solutions, each addressing different aspects of the inference challenge [25, 52].

The fundamental challenge in LLM inference stems from the autoregressive nature of text generation, where each token depends on all previously generated tokens [50]. This sequential dependency limits the parallelization used in traditional batch processing systems [23, 58]. Early inference systems treated this sequential constraint as immutable, leading to suboptimal resource utilization and poor scalability characteristics.

The breakthrough came with the recognition that while token generation must remain sequential, the underlying computations could be parallelized through sophisticated attention mechanisms and memory management techniques [12, 13]. This insight led to the development of techniques such as continuous batching [59] which allows dynamic request scheduling without compromising generation quality. However, memory management represents one of the most critical challenges in LLM inference systems [19, 21, 60]. The Key-Value (KV) cache, essential for efficient attention computation, grows dynamically during generation and can consume substantial memory resources [31, 36]. Traditional memory allocation strategies, designed for fixed-size allocations, prove inadequate for the dynamic and variable-size memory requirements of LLM

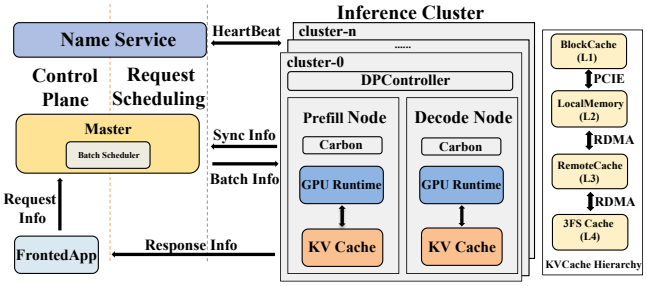


Figure 1: RTP-LLM System Architecture

inference. To address these challenges, paged memory management systems, such as PagedAttention [30], have emerged as a revolutionary approach, treating KV cache as a collection of fixed-size pages that can be allocated, deallocated, and shared across different requests, thereby enabling efficient memory management for variable-length sequences and significantly improving memory efficiency and utilization.

Major cloud providers have developed proprietary inference platforms optimized for their specific infrastructure and hardware configurations [30, 40, 64]. These platforms often achieve impressive performance metrics through extensive optimization and specialized hardware, but suffer from vendor lock-in and limited customization capabilities. The open source ecosystem has produced several notable inference engines, with vLLM [30] representing a significant advancement in the field. However, vLLM and similar systems face limitations in production environments, particularly in their focus on single-node optimization and limited support for heterogeneous hardware environments.

The LLM inference process can be decomposed into two distinct phases with different computational characteristics. The **Prefill Phase** processes the entire input prompt in parallel, generating and storing Key-Value (KV) Cache entries for all prompt tokens, and produces the first output token. This phase involves parallel computation across all input tokens, making it compute-bound. The **Decode Phase** generates subsequent tokens autoregressively, utilizing the current token and historical KV Cache. Each decode iteration processes a single token and updates the KV Cache accordingly, making it memory bandwidth-bound. **Prefill-Decode Disaggregation** exploits this computational asymmetry by physically decoupling these two phases onto dedicated computational resources, enabling independent scaling: prefill engines maximize throughput via large batches while decode engines optimize for low-latency memory access [41, 43, 66].

Speculative Decoding addresses the sequential decode bottleneck by introducing parallel token verification mechanisms [46]. The approach employs a dual-model architecture: a **Propose Model (M_q)** that generates k candidate tokens, and a **Score Model (M_p)** that validates proposed tokens through parallel scoring. The process operates through three stages: (1) Propose stage generates k candidates, (2) Score stage evaluates all k tokens in parallel, and (3) Verification stage determines acceptance based on probability distributions. This transforms sequential decode into parallel verification, significantly improving GPU utilization. For single-request scenarios with negligible verification overhead, when using a rule-based propose model, the theoretical speedup is proportional to the ratio of scoring k tokens versus sequential decoding, with empirical results demonstrating substantial speedup in ideal conditions. As request concurrency increases, speculative sampling's latency benefits diminish due to computational overhead from rejected tokens and resource contention. However, it remains beneficial in memory-constrained environments, scenarios with framework limitations preventing maximum concurrency, and long sequence optimizations requiring improved KV Cache access patterns.

Production LLM deployments require high scalability. The dynamic nature of LLM workloads, characterized by high variability in input length, output length, and computational requirements, resulting in complex scheduling and resource allocation. Modern production environments often involve diverse hardware configurations, from high-end NVIDIA GPUs to AMD ROCm systems and custom accelerators, each requiring specialized implementations for optimal performance. The complexity of LLM inference optimization requires comprehensive solutions that integrate multiple optimization techniques into cohesive systems.

RTP-LLM represents such a comprehensive solution, developed through extensive production deployment experience across Alibaba's ecosystem and designed to address the multifaceted challenges of LLM inference optimization.

3 RTP-LLM System Design

In this section, we present a high-level overview of the comprehensive architectural design of the RTP-LLM inference framework.

3.1 Core System Components

Figure 1 illustrates the main components of RTP-LLM, which include the Frontend Application, the Master, the Prefill Node, the Decode Node, a Multi-Tiered Cache, the Name Service, and the DP-Controller.

The Frontend Application serves as the entry point, tasked with accepting user requests and subsequently returning the corresponding responses. It handles request preprocessing, including tokenization and metadata extraction, before forwarding requests to the Master component for scheduling and execution. The Name-Service performs heartbeat detection and service discovery to ascertain which deployed clusters are operational. It is not responsible for load balancing. Instead, the Master component assumes the critical role of traffic scheduling and global coordination. The Master maintains a global view of system state, including worker availability, KV Cache distribution, and current load conditions, enabling optimal scheduling decisions that maximize throughput while meeting latency requirements.

Regarding practical deployment, our framework supports two distinct strategies: PD-Fusion and PD-Disaggregation. The former, PD-Fusion, refers to a deployment mode where both the prefill and decode phases of the inference process are co-located and executed within a single, unified node. In contrast, the latter, PD-Disaggregation—which is the topology depicted in Figure 1—physically decouples these two computational stages into dedicated nodes. Each operational inference service node is accompanied by a dedicated Carbon service, which is responsible for the automatic recovery and restart of the inference service in the event of a failure.

Algorithm 1: RTP-LLM Hierarchical Architecture

Input: User request *req*, Cluster state *cluster_state*
Output: Processing result *result*

```
// Generate hash keys for prefix matching
1  ℋ ← GenerateHashKeys(req.tokens)
// Find matched blocks in cache
2  ℳ ← PrefixMatching(ℋ)
// Master load balancing and batching
3  ℬ ← MasterDecision(req, cluster_state)
4  forall req ∈ ℬ do
5      if req.kv_cache ∈ block_cache then
           // BlockCache layer: GPU memory
6          UpdateReferenceCount(req.kv_cache)
7      else if req.kv_cache ∈ local_memory then
           // LocalMemory layer: Local CPU memory
8          LoadToGPU(req.kv_cache)
9      else if req.kv_cache ∈ remote_cache then
           // RemoteMemory layer: Remote CPU memory
10         RDMATransfer(req.kv_cache, local_memory)
11     else
           // Remote3fs layer: Distributed storage
12         LoadFrom3FS(req.kv_cache, remote_cache)
// Execute inference on batch
13 result ← ExecuteInference(ℬ)
// Return cache and update LRU
14 CacheReturnAndUpdate(ℬ.kv_cache)
15 return result
```

The Multi-Tiered Cache is a specialized component architected to enhance cache utilization efficiency throughout the inference pipeline. It implements a hierarchical storage system spanning GPU memory, local CPU memory, remote CPU memory via RDMA, and distributed storage, enabling efficient KV cache reuse across requests while minimizing memory bandwidth pressure.

Finally, the DP-Controller is responsible for orchestrating and managing the execution of a batch of requests within a singular deployment context. It coordinates with the Master to receive scheduled batches and manages local resource allocation, including GPU memory management and batch execution. The Prefill Node and Decode Node represent the core inference execution components. Prefill Nodes handle the compute-intensive prefill phase, processing entire input prompts in parallel and generating initial KV cache states. Decode Nodes manage the memory-bound decode phase, generating tokens autoregressively using the cached attention states. In PD-Disaggregation deployments, these nodes operate independently, allowing for specialized optimization and independent scaling of each phase.

3.2 Execution Stream

The complete system workflow of RTP-LLM is outlined in the pseudocode provided in Algorithm 1. Initial user requests are directed to one of the available FrontedApp instances (which operates in a load-balanced configuration). Each FrontedApp performs initial request preprocessing and metadata extraction before

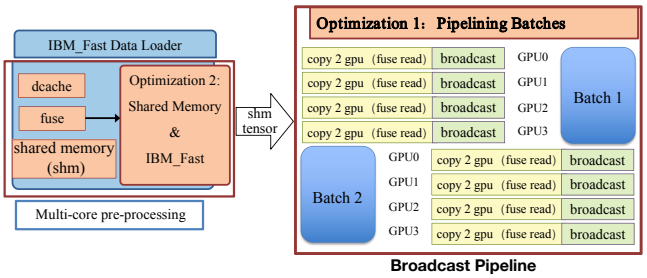


Figure 2: Model Load Optimizations

synchronously forwarding the comprehensive request payload to the centralized Master node. **The Master Node initiates the request processing workflow** by generating the requisite **prefix hash keys ($\mathcal{H}$)** from the incoming user request (Algorithm 1, Line 1: `GenerateHashKeys`). The Master node then utilizes these generated hash keys ($\mathcal{H}$) to perform **prefix matching** against the global cache, identifying the set of matched candidate KV cache blocks ($\mathcal{M}$) (Algorithm 1, Line 2: `PrefixMatching`). Next, the Master node performs decisive **load analysis and scheduling** based on the request details, the matching results ($\mathcal{M}$), and the current `cluster_state`, formulating an execution batch ($\mathcal{B}$) (Algorithm 1, Line 3: `MasterDecision`). This batch is subsequently dispatched to the designated Inference Node for execution.

The core of the system's efficiency is implemented through a **four-tier hierarchical memory access mechanism** (Algorithm 1, Lines 4–12). For every request $req \in \mathcal{B}$, the system attempts to load the necessary KV cache blocks (`req.kv_cache`) from the fastest to the slowest storage tiers:

(1) **GPU Memory (`block_cache`):** If the block is present, its reference count is updated (`UpdateReferenceCount`).
(2) **Local CPU Memory (`local_memory`):** If absent from the GPU, the block is loaded from the local memory to the GPU (`LoadToGPU`).
(3) **Remote CPU Memory (`remote_cache`):** If absent locally, the block is transferred from remote CPU memory to local CPU memory using high-speed **RDMA** (`RDMATransfer`).
(4) **Distributed Storage (`Remote3fs`):** If the block is not found in any higher-tier memory, it is retrieved from the centralized distributed storage and placed into the remote cache (`LoadFrom3FS`).

This hierarchical memory access mechanism ensures that all KV cache dependencies are fulfilled and staged onto the GPU before the inference step. After that, the Inference Node performs the **model execution** on the batch ($\mathcal{B}$) (Algorithm 1, Line 13: `ExecuteInference`). Upon completion, the utilized KV cache blocks are returned and their metadata updated via `CacheReturnAndUpdate` (Algorithm 1, Line 14), which specifically includes the critical step of updating the **Least Recently Used (LRU)** metrics for effective cache management across all memory tiers. The final result is then returned (Algorithm 1, Line 15).

The Inference Node subsequently delivers the final result directly back to the originating FrontedApp to complete the request cycle.

4 Efficient Model Loading

LLM loading presents significant performance challenges in distributed environments, particularly when dealing with models exceeding hundreds of billions of parameters across multiple tensor

parallel processes. Critically, in the Alibaba production environment, all model checkpoints are stored on an internal FUSE (Filesystem in Userspace) cloud storage system, which is mounted on the inference nodes. This infrastructure imposes a strict constraint: the efficiency of file access is heavily dependent on I/O patterns.

For traditional model-structure-driven loading approach, each tensor parallel process reads all model files to extract only its assigned tensor portions. This creates two critical performance problems: redundant file reads across processes and non-sequential file access patterns that severely degrade FUSE prefetching efficiency and file-level caching utilization.

RTP-LLM addresses the above performance problems through a comprehensive optimization strategy that tackles both I/O efficiency and memory management bottlenecks. Our strategy centers on restructuring the loading paradigm from model-structure-driven to file-order-driven loading. Instead of iterating through weight modules and loading their constituent tensors, we iterate through model files sequentially, loading all tensors from each file before proceeding to the next. This approach ensures sequential file access patterns that maximize FUSE prefetching effectiveness while maintaining compatibility with community engines.

Figure 2 highlights our key optimizations. To eliminate redundant reads, we integrate IBM's fastsafetensors library [57] with RTP fastsafetensors implementation. The hybrid approach assigns each model file to a single process for reading, then leverages PyTorch distributed broadcast to efficiently share tensors across all processes. This eliminates the need for each process to read every file while maintaining the benefits of FUSE-based I/O optimizations.

Memory management optimization reduces the significant overhead of repeated shared memory allocation. The original fastsafetensors library allocates and registers pinned memory for each file read operation, requiring 600ms overhead per 2GB allocation. We encapsulate the loading interface within a class that reuses a single shared memory buffer across multiple file reads, eliminating this redundant allocation cost.

Finally, we optimize communication and I/O overlap by parallelizing file reading operations with tensor broadcasting. Rather than sequentially reading files followed by tensor distribution, we overlap these two phases (i.e., parallelizing file reading and tensor distribution) in order to reduce overall loading latency. This is achieved by allowing file I/O operations to proceed concurrently with tensor broadcasting, maximizing resource utilization across the distributed system.

The combination of file-ordered loading, hybrid distributed reading, shared memory reuse, and parallel I/O-communication overlapping creates a comprehensive solution that improves the performance of large-scale model loading while maintaining compatibility with existing model formats and community frameworks.

5 Traffic Scheduling

In this section, we outline the traffic scheduling mechanism. We describe the Load Balance Strategy of RTP-LLM, which dynamically redistributes requests based on real-time workloads. Next, we introduce KV cache Management to enable efficient memory utilization and consistent serving performance across workers.

5.1 Load Balancing Strategy

RTP-LLM employs distinct load balancing mechanisms for Prefill and Decode stages to optimize throughput and latency.

For Prefill requests, FrontApp tokenizes input sequences and generates block hash identifiers. Each request is divided into blocks (e.g., 64 tokens per block), and each block's hash key is computed based on its token IDs. FrontApp sends to Master the request's block hash IDs, sequence length, and optional chat ID.

Master groups requests with similar sequence lengths into batches to minimize padding overhead. Window size $w = \max(DP\_size, |R|)$ is dynamically adjusted based on DP group size and queue depth. Master queries DP-Controllers for real-time load status, including running/waiting requests, GPU memory, and KV cache occupancy.

When all DP-Controllers are busy, Master employs predictive scheduling by estimating completion times:

$$t_{available}(d_i) = \max_{r \in running(d_i)} t_{start}(r) + \hat{t}_{prefill}(r) \tag{1}$$

where $t_{start}(r)$ denotes the start time when request r began execution on DP-Controller d_i, and $\hat{t}_{prefill}(r)$ is predicted based on sequence length and batch composition. Master schedules requests to the DP-Controller expected to complete first, in order to reduce the queue wait time.

Decode requests prioritize KV cache affinity: when a request arrives with a chat ID, Master checks if this chat session was previously assigned to a worker. If a match exists and the worker has sufficient capacity, Master directly routes to that worker, exploiting local KV cache locality. For cache management, Master implements admission control, eviction priority, and backpressure signaling to prevent cache thrashing.

5.2 KV Cache Management

RTP-LLM employs a hash-based approach for efficient prefix matching across workers. The Local KV Cache Manager maintains a unified hash map that aggregates cache keys from all workers, mapping hash keys to block identifiers and worker metadata.

5.2.1 Local KV Cache Manager. The Local KV Cache Manager aggregates cache keys from all workers into a unified hash map structure. Each entry in the map stores:

- **Hash key**: Block hash identifier computed from token IDs in the block
- **Block ID**: Block identifier on the worker
- **Worker cache info**: Set of $(worker\_id, cache\_metadata)$ pairs indicating which workers have cached this block

Instead of maintaining separate hash maps for each worker requiring $O(B \times W)$ lookups, we merge cache keys from all workers into a single hash map, enabling $O(B)$ complexity for prefix matching, where B is the number of blocks and W is the number of workers.

Master queries worker status at high frequency (20ms) for scheduling decisions, while cache key synchronization operates at lower frequency (50ms). Workers maintain cache version numbers. When requesting cache keys, the manager includes the last known version. If unchanged, workers return lightweight acknowledgment. If changed, workers return delta updates to minimize data transfer.

Algorithm 2: Prefix Cache Matching

Input: Block hash IDs $H = [h_1, h_2, \ldots, h_B]$, Unified hash map $\mathcal{H}$

Output: Mapping $M : WorkerID \rightarrow MatchLength$

```
l ← 0;
M ← ∅;
for i = 1 to B do
    if h_i ∈ H then
        worker_info ← H[h_i]; // Get workers
        l ← l + 1;
        forall w ∈ worker_info do
            M[w] ← max(M[w], l); // Update max length
    else
        break // Terminate
return M
```

5.2.2 Prefix Cache Matching. When FrontApp sends a request with block hash IDs $H = [h_1, h_2, \ldots, h_B]$ (computed from token sequences), Master queries the Local KV Cache Manager for prefix matching. The matching process queries the unified hash map:

This single-pass hash map lookup enables efficient matching: rather than querying each worker's hash map separately ($O(B \times W)$), we query the unified hash map once per block ($O(B)$), aggregating match results across all workers simultaneously.

5.2.3 Sampled Prefix Hashing. For prefix matching on worker nodes, workers use sampled prefix hashing to balance matching granularity with storage overhead. When a cached block contains fewer tokens than a threshold (e.g., 208 tokens), only that length is hashed. For larger blocks, multiple hash entries are created at regular intervals.

Specifically, for a block with $n \geq 208$ tokens, hash entries are created at positions: 208, 212, 216, 220, 224, 228, ..., up to n. This sampling strategy, parameterized by start threshold (208) and step size (4), enables prefix matching at multiple granularities while controlling metadata overhead.

When prefix matching occurs, the system matches against all sampled hash positions for each block. Matched blocks are categorized as either full (all tokens cached) or partially-filled (watermark indicates available space). Full blocks use reference counting for concurrent access by multiple requests, while partially-filled blocks are exclusive and allow requests to append tokens directly after the watermark.

5.2.4 Remote KV Cache Manager Server. The Remote KV Cache Manager Server is deployed as per-datacenter instances for capacity management and fault isolation. Unlike the hash-based Local Manager, it maintains a simple mapping structure: *cache key* $\rightarrow$ *file path*, optimized for persistent storage lookups on 3FS. It provides durability guarantees through persistent metadata storage, enabling cache recovery after system restarts.

5.2.5 Cache Matching and Scheduling Integration. When Master receives a scheduling request, it performs parallel lookups to both Local and Remote KV Cache Managers:

- **Local Cache Query**: Query Local KV Cache Manager's unified hash map for worker-level cache matches, returning maximum match length per worker
- **Remote Cache Query**: Query Remote KV Cache Manager Server for 3FS cache matches, returning maximum match length from persistent storage

Both queries execute concurrently. Master combines results to compute the cache reuse score for each candidate worker w:

$$\begin{aligned} score(w) =& \alpha \cdot \frac{local\_match\_len(w)}{total\_seq\_len} \\ &+ \beta \cdot \frac{remote\_match\_len}{total\_seq\_len} \\ &- \gamma \cdot \frac{predicted\_latency(w)}{max\_latency} \end{aligned} \tag{2}$$

where α, β, and γ are weighting factors tuned based on workload characteristics. This score is combined with worker load information to make final scheduling decisions.

If the request includes a chat ID, Master uses it as a strong hint: if the chat session was previously assigned to a worker with cached KV state, Master prioritizes routing to that worker, enabling direct reuse of local cache.

6 Speculative Decoding Design

This section presents RTP-LLM's speculative sampling framework, which enables Aone Copilot to achieve 1000 tokens per second inference performance in production deployments [5]. We examine the theoretical foundations, implementation architecture, and performance characteristics of speculative sampling in large-scale LLM deployment.

6.1 RTP-LLM Speculative Sampling Framework

The RTP-LLM framework implements a comprehensive speculative sampling system designed to support multiple speculative sampling algorithms while maintaining modularity and extensibility. The framework's underlying implementation is built in C++.

6.1.1 Architecture Overview. The framework decomposes speculative sampling into four modular components: i)**ProposeExecutor**: Manages token proposal generation across different algorithms (naive speculative sampling, Prompt Lookup, Eagle, MTP); ii) **ScoreExecutor**: Handles parallel token scoring by the target model; iii) **SpeculativeSampler**: Implements verification algorithms to determine token acceptance; and iv) **SpeculativeUpdater**: Updates accepted tokens to the original stream.

The execution flow is as below: (1) **ProposeExecutor** generates k candidate tokens using the configured proposal algorithm; (2) **ScoreExecutor** performs parallel forward passes through the target model to score all k candidate tokens simultaneously; (3) **SpeculativeSampler** applies verification algorithms (e.g., standard speculative sampling acceptance criteria) to determine which candidate tokens to accept based on the probability distributions; (4) **SpeculativeUpdater** integrates the accepted tokens into the original generation stream, advancing the generation state accordingly. Each component maintains clear input/output interfaces and stateless operation, ensuring loose coupling and facilitating algorithm experimentation.

Algorithm 3: N-gram Token Matching and Speculative Sampling

Input: Input prompt *prompt*, Recently generated tokens *recent_tokens*, Proposal count k
Output: Accepted tokens *accepted_tokens*

```
// Match n-grams
match_result ← MatchNGrams(prompt, recent_tokens);
if match_result succeeds then
    // Extract k candidates
    candidates ← ExtractTokens(match_result, k);
    // Validate
    validation_results ← ScoreModel(candidates);
    // Verify and accept
    accepted_tokens ← VerificationAlgorithm(candidates,
      validation_results) ;
return accepted_tokens
```

6.1.2 Supported Algorithms. The framework supports multiple speculative sampling approaches: i) **Naive Speculative Sampling**: Direct use of smaller GPT models as propose models; ii) **MTP (Multi-Token Prediction)** [14, 54]: Predicts multiple next tokens in one forward pass for parallel verification (e.g., DeepSeek-V3); iii) **Eagle** [33]: Novel Auto-Regression Head training for future hidden state prediction; iv) **Prompt Lookup**: N-gram matching against historical prompts for token proposal.

6.2 Prompt Lookup Speculative Sampling

Prompt Lookup represents a specialized form of speculative sampling particularly effective for extractive scenarios, where generated content can be directly copied from input prompts. The algorithm operates through n-gram token matching against the input prompt using recently generated tokens, extracting subsequent k tokens as candidate proposals, and validating them through the score model. Algorithm 3 presents the complete procedure.

For code editing applications [5], Prompt Lookup benefits from additional optimizations: **cursor maintenance** to track the last successful lookup position ensuring sequential copying, **skip initial matching** to use the first k tokens directly in the initial iteration, and **position updates** to advance cursor position after each successful iteration. These optimizations leverage the sequential nature of code copying, where subsequent operations continue from previously copied positions.

7 Runtime and System Extensions

RTP-LLM also incorporates a series of runtime and system-level optimizations for efficient large-scale inference, including **Parallel Execution**, **Quantization**, and **Multimodal Model Support**.

7.1 Parallel Execution

The enormous parameter count of state-of-the-art Large Language Models (LLMs), often exceeding hundreds of billions (e.g., 96B), necessitates sophisticated **multi-level parallelism strategies** to distribute computational load and memory requirements across multiple nodes and devices. Effective parallelism is critical for minimizing the Time-To-First-Token (TTFT) and maximizing system throughput in production environments. We employ a hierarchical parallelism framework integrating Tensor Parallelism, Pipeline Parallelism, Data Parallelism, and Expert Parallelism.

- **Tensor Parallelism (TP)**: TP is applied to individual weight matrices (e.g., in FFN and Attention layers) within a transformer block, partitioning them across multiple GPUs within a node. This technique addresses the single-GPU memory limit for the largest models and is crucial for accelerating the compute-bound operations (e.g., matrix multiplications) during the **Prefill phase** (context processing). High-speed interconnects like NVLink are essential for efficient inter-GPU communication during all-gather and reduce-scatter operations.
- **Pipeline Parallelism (PP)**: PP distributes consecutive layers of the model across a sequence of GPUs, forming a pipeline. PP is essential for handling models that cannot fit even with TP on a single node. PP introduces **pipeline bubbles**, which can be mitigated by micro-batching.
- **Data Parallelism (DP)**: DP is applied at the cluster level, replicating the model weights across multiple nodes. It is primarily used to scale throughput by processing multiple batches of requests simultaneously. This is combined with sophisticated dynamic batching and load balancing to maintain high GPU utilization.
- **Expert Parallelism (EP)**: EP is designed for **Mixture-of-Experts (MoE)** models. The sparse FFN experts are distributed across multiple devices, and during inference, the routing mechanism only activates a small subset of experts. EP is memory-efficient and good for scaling MoE models, though it increases complexity in load balancing and dynamic routing communication.

7.2 Quantization Techniques

Model quantization is an indispensable optimization for LLM inference, addressing the critical challenges of memory capacity and bandwidth saturation. Reducing the precision of model weights and intermediate states from FP16/BF16 to lower-bit formats (e.g., INT8/INT4) directly enhances throughput and enables the deployment of larger models on commodity hardware.

7.2.1 Weight-Only Quantization. As Model weights constitute the majority of the model's memory footprint, we primarily employ **Weight-Only Quantization** methods. These techniques convert the weights to lower precision (typically INT4/INT8) while keeping the activations in higher precision for computation (e.g., FP16/BF16) to minimize quality loss. State-of-the-art methods include:

- **GPTQ (General Pipelined Token Quantization)** [18]: As an efficient, one-shot Post-Training Quantization (PTQ) method, GPTQ remains foundational for its fast implementation. However, industry focus is shifting towards more robust frameworks.
- **AQT (Activation-aware Quantization) Frameworks**: Modern approaches such as **AWQ** (Activation-aware Weight Quantization) [34] and **HQQ** (Half-Quadratic Quantization) [6] are preferred for pushing to extremely low bitwidths (e.g., INT3, INT2) by leveraging activation distributions or advanced non-linear optimization techniques, ensuring minimal degradation in LLM reasoning capability.

Furthermore, the emerging industry standard of **FP8 (8-bit Floating Point)** is integrated, offering high performance and minimal

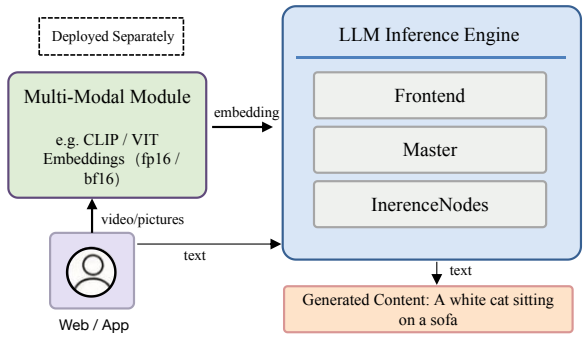


Figure 3: EPD Disaggregation

accuracy degradation, particularly when combined with hardware accelerators supporting the format.

7.2.2 KV Cache Quantization. The Key-Value (KV) cache, which stores intermediate attention states, dynamically grows with context length and quickly becomes the bottleneck in memory bandwidth and capacity, especially for models supporting contexts of 128K+ tokens. To mitigate this memory pressure, we employ standard quantization techniques applied specifically to the KV Cache.

- **On-the-fly Quantization**: The Key and Value tensors are quantized from FP16/BF16 to lower precision (typically INT8, INT4 or FP8) during the generation. It directly uses **per-tensor or per-block dynamic scaling** to determine the quantization factor, prioritizing hardware efficiency and speed.
- **Memory Footprint Reduction**: Quantizing the KV cache effectively reduces its size by half (for INT8/FP8) or more. This directly alleviates the memory bandwidth pressure associated with the Decode Phase. This is essential for maximizing the effective batch size and concurrent request capacity of the engine under memory-bound conditions.

This memory-centric optimization aims to maximize concurrent request capacity and support extremely long-context workloads.

7.3 Multimodal Model Support

Multimodal models represent a significant advancement in artificial intelligence, enabling communication through multiple modalities with computers. These models aim to process and understand multimodal information, including images, videos, and audio. In the context of RTP-LLM, our focus is primarily on models that accept images as input, specifically supporting prominent multimodal architectures such as LLaVA [37] and Qwen-VL [8].

7.3.1 Supported Multimodal Architectures. LLaVA Architecture: The LLaVA (Large Language and Vision Assistant) model integration follows the HuggingFace format specification. The configuration file (`config.json`) contains the `mm_vision_tower` keyword, which specifies the path to the Vision Transformer (ViT) component [16]. Typically, this implementation utilizes OpenAI's pre-trained CLIP model for visual feature extraction.

Invocation Interface: The invocation mechanism maintains consistency with the HuggingFace format. Users specify image insertion positions using the `<image>` tag within the prompt. Images are provided as a sequence in `List[str]` format. Notably, RTP-LLM's multimodal interface supports inserting multiple images in a single prompt, though the effectiveness of current supported models on multiple images remains limited. It is crucial to ensure strict correspondence between the number of image tags and the number of provided images.

Qwen-VL Architecture: The Qwen-VL implementation differs slightly from LLaVA in its architectural approach. While Qwen-VL's ViT component also employs CLIP, its parameters are integrated with the Large Language Model (LLM) portion, resulting in ViT parameters being read directly from the model checkpoint rather than from separate configuration files.

Invocation Interface: Similar to LLaVA, Qwen-VL follows HuggingFace format conventions. Images are marked using the `<img>{img_url}</img>` tag within prompts. Additionally, the system supports `<img/>` placeholder syntax, enabling separation between URL specification and prompt input for enhanced flexibility.

7.3.2 Service Deployment. For production use, RTP-LLM supports server-based multimodal model serving. In general, users provide the model with video or image inputs, along with corresponding textual input. The visual data (video or image) is first fed into the ViT model, which generates embedding data. This generated embedding data is then concatenated with the textual input and provided to the Language Model (LM) to ultimately produce the final output. In Figure 3, we present the actual deployment architecture for our Vision Transformer (ViT) model. We adopted a decoupled, independent deployment strategy for the Large Language Model (LLM) and the multimodal model [45]. This architecture enables ViT models and language models to utilize separate streams during inference, avoiding contention between them. This design allows for computation overlap under high request concurrency, thereby improving overall performance. Additionally, this design provides a hidden benefit by reducing the actual GPU memory footprint on a single device.

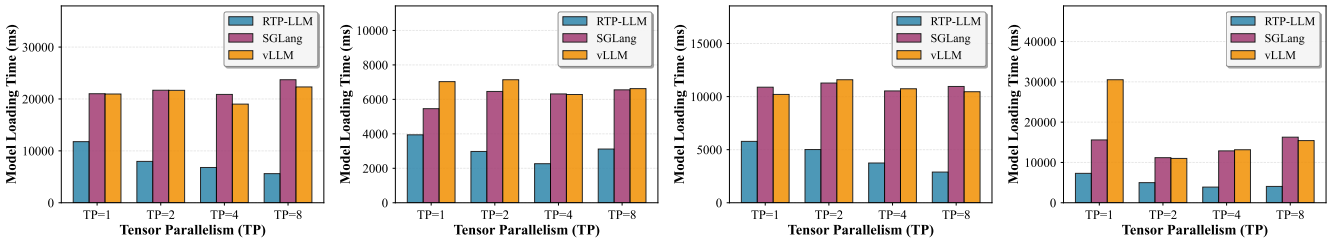
This multimodal support framework provides a robust foundation for integrating vision-language capabilities into large-scale language model deployments, addressing the growing demand for multimodal AI applications in industrial settings.

8 Experiments

To comprehensively validate the superiority and effectiveness of the RTP-LLM inference framework, we present a set of evaluations under various LLMs and real production environments, and compare its performance against two prominent state-of-the-art open-source frameworks: **vLLM** [30] and **SGLang** [64].

We **prioritize evaluations that use real business traffic** on traffic scheduling, PD disaggregation, and speculative decoding; then the evaluations use controlled benchmarks on model loading, quantization, and multimodal (ViT/EPD). **For comparisons of RTP-LLM with SGLang and vLLM on production traffic** (PD Disaggregation and Speculative Sampling), the workload is **input 200K tokens, output 16K tokens** throughout. All evaluations are conducted on servers running Linux kernel version 5.10 (x86_64), each equipped with 64 CPU cores, 600GB memory, and 8 GPUs.

Table 1 presents a summary of the core metrics employed in our evaluations to quantify the performance and industrial readiness of the RTP-LLM framework. The evaluation is structured as follows.



(a) Qwen3-32B (b) Qwen3-8B (c) Qwen2.5-14B-Instruct (d) Moonlight-16B-A3B

Figure 4: Model loading time comparison for medium-scale models (8B-32B parameters) across different TP configurations.

(1) **Production Traffic Scheduling Effectiveness (addressing Challenge I and II):** Validating the impact and stability of traffic scheduling with KV cache management under **real deployment environments**, demonstrating GPU utilization and memory efficiency in production.
(2) **Deployment Evaluation of PD Disaggregation (addressing Challenge I and III):** Evaluating Prefill-Decode Disaggregation with large-scale MoE models using **real business traffic** from actual online deployments, demonstrating GPU utilization and architectural flexibility for heterogeneous model types.
(3) **Speculative Decoding Performance Gain (addressing Challenge I):** Quantifying the acceleration and throughput improvement. We further report one **real deployment scenario** for **user code editing prompt generation**.
(4) **Model Loading Superiority (addressing Challenge IV):** Benchmarking loading latency against SOTA baselines to confirm superiority for rapid elastic scaling and model iteration.
(5) **Quantized Inference Efficiency (addressing Challenge II):** Assessing throughput and memory footprint improvements from advanced quantization and KV cache management, demonstrating memory efficiency gains.
(6) **Multimodal Inference Performance (addressing Challenge III):** Evaluating end-to-end efficiency of complex multimodal models via decoupled ViT and LLM (EPD) processing: the public GQA benchmark for framework comparison, and **real production deployment** for RTP-LLM RT and TTFT across concurrency, demonstrating architectural flexibility in production.

8.1 Traffic Scheduling Strategies Validation

We use real deployment environments within Alibaba: **internal robot Q&A service** and **Taobao merchant customer service consultation**. The reported metrics are TTFT P95, inference P95, and cache reuse length. We evaluate RTP-LLM's traffic scheduling with KV cache management, comparing against baseline strategies without traffic scheduling.

Table 2 presents the latency comparison. For **internal robot Q&A service** (Qwen 7B model, input length 300–1000 tokens, average 340 tokens, output length 6–7 tokens), RTP-LLM with traffic scheduling achieves significant improvements: TTFT P95 latency reduces from 83.3 ms to 52.3 ms (37.2% reduction), while maintaining similar average inference latency; the improved cache reuse (Table 3) enables 75% reduction in prefill machine count (from 80 to 20 machines) while maintaining average TTFT performance. For **Taobao merchant customer service consultation** (Qwen 4B model, input length 2500 tokens, output length 114 tokens), RTP-LLM with traffic scheduling also demonstrates consistent performance improvements compared to that without traffic scheduling.

Table 1: Key metrics for performance evaluation

Metric	Unit	Description
TTFT	ms	Time to first token
Tokens/s	T/s	System throughput
GPU Memory	MB	Peak Runtime Memory Usage
Cache Hit Rate	%	Efficiency of KV cache management
Batch Latency	ms	Total cost time for batch execution
PPL	N/A	Model output fidelity

Table 2: Traffic scheduling performance comparison across two real production workloads. TS: Traffic Scheduling. Latency is measured in milliseconds (ms).

Workload	TS	TTFT P95	Inference P95
Internal Robot Q&A	Off	83.3	136
	On	52.3	96.2
Taobao Merchant Service	Off	350	1760
	On	226	1210

Table 3: KV cache reuse length comparison across different strategies. Cache reuse length is measured in tokens.

Workload	TS	Cache Reuse Length
Internal Robot Q&A	Off	26.6
	On	83.8
Taobao Merchant Service	Off	833
	On	840

RTP-LLM reduces TTFT P95 latency from 350 ms to 226 ms (35.4% reduction), and inference P95 latency from 1760 ms to 1210 ms (31.3% reduction).

Table 3 summarizes cache reuse. RTP-LLM's unified hash map and cache affinity-based routing improve reuse length: for internal robot Q&A, from 26.6 to 83.8 tokens (215% improvement); for Taobao merchant service, from 833 to 840 tokens. This reduces prefill overhead and latency.

8.2 PD Disaggregation Performance

We evaluate RTP-LLM's Prefill-Decode Disaggregation architecture using the Qwen3-Coder-480B-FP8 model [49], a large-scale MoE (Mixture-of-Experts) model with FP8 quantization. Note that, Qwen3-Coder-480B-FP8 is deployed in online business environment. The workload is configured with a batch size of 64 to match production traffic patterns.

Table 4: Performance comparison for Qwen3-Coder-480B-A35B-Instruct-FP8 (KV Cache FP8 enabled) across different frameworks. (RTP-LLM speedup: Cache Hit Rate: SGLang 1.57x, vLLM 2.36x; TTFT: SGLang 4.72x, vLLM 5.33x)

Metric	RTP-LLM	SGLang	vLLM
Cache Hit Rate (%)	45.09	28.70	19.10
TTFT (ms)	1338.38	6322.7	7134.8
Tokens/s	1081.72	1152.95	1084.58

Deployment Configuration: The deployment employs a distributed setup across 5 nodes, each equipped with 8 GPUs, implementing the Prefill-Decode Disaggregation architecture. Specifically, 4 nodes are dedicated to Prefill processing, while 1 node handles Decode operations. This asymmetric allocation reflects the typical workload characteristics where prefill requires more computational resources due to parallel processing of input sequences, while decode benefits from higher concurrency with lower per-request computational intensity.

Parallelism and Communication: Each node is configured with Tensor Parallelism (TP=8), Expert Parallelism (EP=8), and Data Parallelism (DP=1). The Expert Parallelism configuration uses DeepEP for efficient All2All communication among expert layers, optimizing the communication overhead in MoE models [62]. For KV cache transmission between Prefill and Decode nodes in the disaggregated architecture, the system utilizes NCCL IBRC (InfiniBand Reliable Connection) [11] for high-performance, low-latency data transfer, ensuring efficient communication of cached key-value states across the distributed deployment. Prefill nodes operate in normal mode, prioritizing throughput for batch processing, while the Decode node operates in low latency mode, minimizing inter-token latency for real-time generation.

Concurrency Settings: To maximize resource utilization while maintaining latency requirements, Prefill nodes are configured with a batch size of 64, allowing efficient parallel processing of input sequences. The Decode node supports a higher concurrency of 128, capitalizing on the memory-bound nature of decode operations to process multiple requests simultaneously.

Table 4 presents the performance comparison for the Qwen3-Coder-480B-A35B-Instruct-FP8 model with KV Cache FP8 enabled. RTP-LLM achieves a cache hit rate of 45.09%, outperforming SGLang (28.70%) and vLLM (19.10%) by 1.57x and 2.36x, respectively. RTP-LLM achieves the lowest TTFT at 1338.38 ms, demonstrating 4.72x and 5.33x speedup compared to SGLang (6322.7 ms) and vLLM (7134.8 ms). The latency improvement is primarily attributed to traffic scheduling strategies that enable more effective prefix cache reuse. Throughput is essentially consistent across all frameworks: RTP-LLM achieves 1081.72 tokens/s, compared to SGLang (1152.95 tokens/s) and vLLM (1084.58 tokens/s).

8.3 Speculative Sampling Performance

We evaluate RTP-LLM's speculative decoding performance using the DeepSeek-V3-0324 [14] model, where the last layer of the model serves as the draft model for speculative decoding. All runs use Tensor Parallelism (TP=8, DP=1) deployment and consistent parameters: max_batch_size = 32, max_new_tokens = 500, FP8 KV Cache is enabled, and the speculative decoding step size is set to 1 (predicting one token per step).

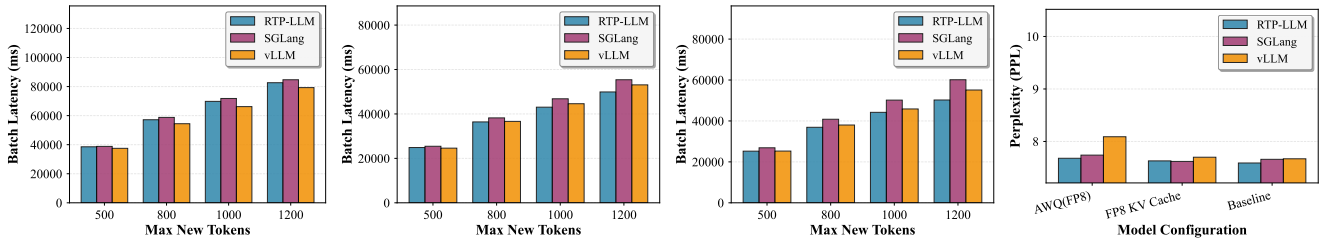
Table 5: Throughput comparison (tokens/s) for DeepSeek-V3-0324 speculative sampling across different frameworks. (RTP-LLM Speedup: compared with vLLM: 1.12x; compared with SGLang: 2.48x)

Framework	RTP-LLM	vLLM	SGLang
Tokens/s	187.53	167.95	75.785

Table 5 presents the throughput comparison in tokens per second across the three frameworks. RTP-LLM achieves the highest throughput at 187.53 tokens/s, demonstrating 1.12x speedup compared to VLLM (167.95 tokens/s) and 2.48x speedup compared to SGLang (75.785 tokens/s). The modest superior performance over vLLM stems from RTP-LLM's direct C++ operator launch mechanism, which eliminates the Python-to-C++ invocation overhead present in open-source frameworks like vLLM that require Python calls to C++ before launching operators, reducing per-operator invocation overhead in the speculative sampling pipeline.

Real Deployment: 235B MoE with MTP (Merchant Data-Agent): To validate speculative decoding under **real production deployment**, we collected metrics using **1,000 real production queries** from an online merchant data-agent service powered by the Qwen3-235B-A22B MoE model [49], with Multi-Token Prediction (MTP) enabled. The workload reflects actual business traffic: **input context cap 20K tokens** (average input length 19.5K tokens, average output length 800 tokens), Prefill 4TP×4 rows with prefix cache reuse, Decode with the same total GPU count. The four decode configurations in Table 6—4TP×4, 1TP8DP, 2TP4DP, and 2TP8DP—are **all used in real production** for this service; we compare them under the same total GPU budget to guide production choice. Table 6 reports decode-side single-card TPS and average time-per-output-token (TPOT) at 64–512 client concurrency.

Deployment Scheme Guidance. In production, teams choose among these schemes according to whether the workload is latency-sensitive (e.g., online interactive) or throughput-oriented (e.g., offline batch). *4TP×4* favors **low concurrency**: it achieves the best TPOT at 64 concurrency (15.83 ms) but TPOT and stability degrade as concurrency grows, and is typical for dedicated low-latency lanes. *1TP8DP* is best for **online, latency-sensitive** serving: it consistently delivers the best TPOT at 128–512 concurrency (17.57–25.64 ms), as higher data parallelism spreads requests and reduces per-request wait; it is the preferred option in our production environment when the goal is to minimize time-per-output-token under real traffic. *2TP4DP* is best for **offline, throughput-oriented** workloads: it achieves the highest decode TPS (e.g., 601.98 at peak) by balancing tensor and data parallelism, maximizing utilization when KV cache is saturated. *2TP8DP* uses more tensor parallelism and exhibits the worst TPOT at high concurrency (e.g., 46.3 ms at 512), as cross-GPU communication under TP becomes the bottleneck; it is still a valid production option when the same cluster must serve both prefill-heavy and decode-heavy phases with a unified TP. Under this real workload, MTP maintains an effective sampling rate of approximately 1.9 tokens per step, with KV cache utilization above 90% and SM utilization around 60%. These results show that RTP-LLM's speculative decoding and the above parallelism choices are validated in **real production environments** where these deployment schemes are in active use.



(a) Batch Latency (AWQ(FP8)) (b) Batch Latency (FP8 KV Cache) (c) Batch Latency (Baseline) (d) Precision Loss (PPL)

Figure 5: Batch latency and precision loss comparison for Qwen3-32B across different quantization configurations.

Table 6: Real deployment (235B MoE + MTP): decode TPS (single card) and average TPOT (ms).

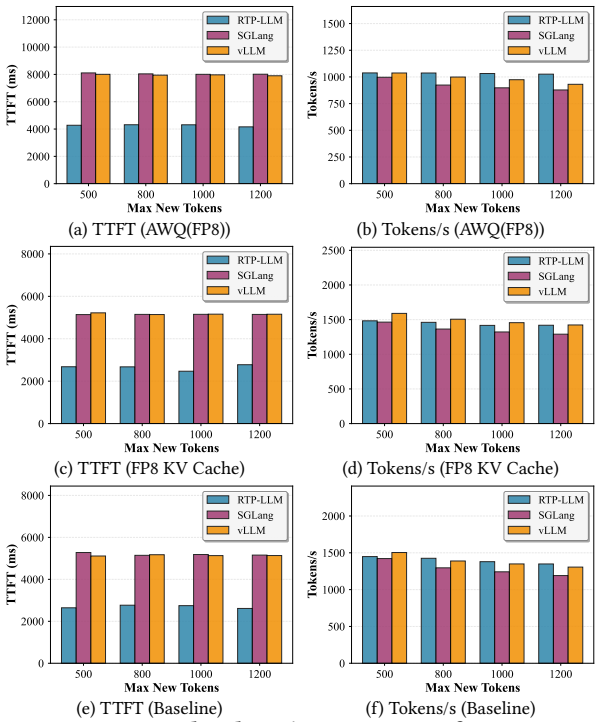
Decode config	64 conc.	128 conc.	256 conc.	512 conc.
Single-card Decode TPS				
4TP×4	147.9	270.9	376.7	427.9
1TP8DP	86.0	202.1	380.8	529.8
2TP4DP	**232.9**	**381.9**	**550.1**	**599.8**
2TP8DP	143.8	262.8	421.0	454.4
Avg. TPOT (ms)				
4TP×4	**15.83**	21.54	27.12	28.4
1TP8DP	17.87	**17.57**	**21.81**	**25.64**
2TP4DP	16.14	19.45	27.07	28.45
2TP8DP	17.21	22.68	35.8	46.3

Table 7: Model loading time comparison for Qwen3-235B-A22B across different TP configurations. Loading times are measured in seconds (s). (RTP-LLM Speedup compared with SGLang and vLLM: TP=4: 4.70x–4.78x; TP=8: 6.18x–6.27x)

TP	RTP-LLM	SGLang	vLLM
TP=4	37.1s	177.4s	174.3s
TP=8	33.0s	206.7s	204.0s

8.4 Model Loading Performance

We evaluate RTP-LLM's model loading performance across five LLM models (8B-235B parameters) [38, 47, 49, 55] under different Tensor Parallelism (TP) configurations, comparing against SGLang and vLLM. Figure 4 shows that RTP-LLM achieves 1.4x-4.2x speedup over baseline frameworks for medium-scale models (8B-32B parameters), with performance improving at higher TP configurations. For the large-scale Qwen3-235B-A22B model [49] (shown in Table 7), RTP-LLM achieves 4.7x-6.3x speedup, reducing loading time from 37.1s (TP=4) to 33.0s (TP=8). This superior performance stems from shared weight loading, where each GPU reads its assigned model partition in parallel, enabling higher TP configurations to achieve more speedup for large-scale models by distributing I/O load. Beyond parallel shared reading, RTP-LLM further optimizes performance by overlapping weight loading with data broadcasting, allowing each GPU to broadcast previously loaded data while simultaneously loading its assigned partition, a capability absent in baseline frameworks. In contrast, SGLang and vLLM exhibit negative scalability, with loading time increasing by 16.5% and 17.0% respectively when scaling from TP=4 to TP=8. The performance advantage becomes more pronounced for larger models and higher TP configurations, indicating superior parallel loading efficiency.



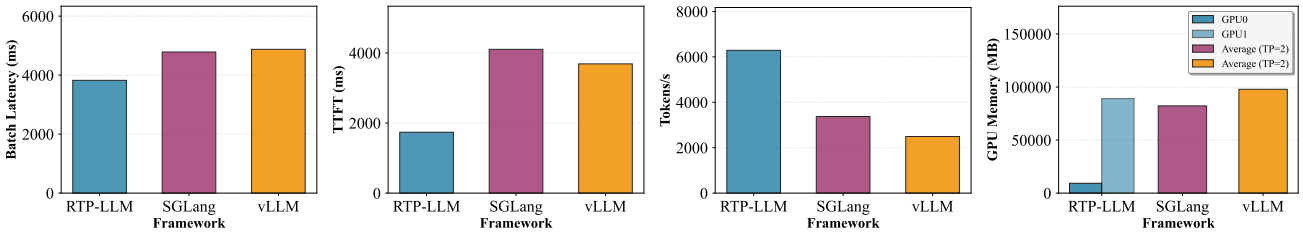
(a) TTFT (AWQ(FP8)) (b) Tokens/s (AWQ(FP8))
(c) TTFT (FP8 KV Cache) (d) Tokens/s (FP8 KV Cache)
(e) TTFT (Baseline) (f) Tokens/s (Baseline)

Figure 6: TTFT and Tokens/s comparison for Qwen3-32B across different quantization configurations.

8.5 Quantized Inference Performance

Throughput and latency are measured under controlled request patterns; precision (PPL) is evaluated on the public WikiText-2 [39] dataset. We evaluate RTP-LLM's quantization performance for Qwen3-32B model across different quantization configurations (AWQ(FP8), FP8 KV Cache, Baseline) [49], comparing against SGLang and vLLM. Here, Baseline denotes Qwen3-32B without KV cache quantization and AWQ quantization. All runs use single GPU deployment (TP = 1, DP = 1) and consistent parameters: max_batch_size = 64, top-p = 1, top-k = 1, temperature = 0.0. The performance metrics include Batch Latency, TTFT, Tokens/s, and PPL on WikiText-2. We vary max_new_tokens over 500, 800, 1000, and 1200.

Figure 5 presents the batch latency and precision loss comparison across different quantization configurations. RTP-LLM demonstrates superior batch latency performance in Baseline configuration (Figure 5(c)), achieving the lowest batch latency across all max_new_tokens settings. For FP8 KV Cache configuration (Figure 5(b)), RTP-LLM achieves the lowest batch latency for longer sequences (800, 1000, 1200 max_new_tokens), demonstrating better scalability compared to baseline frameworks. RTP-LLM achieves



(a) Batch Latency (b) TTFT (c) Tokens/s (d) GPU Memory

Figure 7: Performance and GPU memory utilization comparison for Qwen/Qwen2.5-VL-7B-Instruct on GQA dataset across different frameworks. All frameworks are deployed with TP=2. RTP-LLM shows GPU0 and GPU1 memory usage separately, while SGLang and vLLM show average memory usage across both GPUs.

significant batch latency reduction of 35%-40% in FP8 KV Cache configuration compared to AWQ(FP8) configuration (Figure 5(a)), while maintaining competitive performance in AWQ(FP8) configuration.

Figure 6 presents TTFT and tokens/s results across different configurations. RTP-LLM achieves the lowest TTFT across all the configurations, demonstrating 1.9x-3.0x reduction compared to SGLang and vLLM. In addition, RTP-LLM demonstrates high performance across all the quantization configurations, achieving the highest tokens/s in AWQ(FP8) and Baseline configurations. In FP8 KV Cache configuration, RTP-LLM maintains competitive performance with minimal differences compared to baseline frameworks, while consistently achieving superior batch latency for longer sequences and TTFT. The superior performance is due to RTP-LLM's superior engineering implementation, including optimized quantization kernel and efficient memory access patterns.

Figure 5(d) shows precision loss (PPL) on a sampled subset of WikiText-2. All configurations yield PPL in 7.59–8.09 (baseline 7.59–7.67). RTP-LLM attains the best precision in Baseline and comparable precision in AWQ(FP8) and FP8 KV Cache (within 0.01 PPL of baseline).

8.6 EPD Disaggregation Performance

The evaluation is motivated by production vision-language workloads such as **structured product description generation** (e.g., for second-hand marketplace listings): each request contains **multiple product images** and an **image-retrieval-based reference product list**, requiring multi-image understanding and coherent long-form text generation. To enable reproducible framework comparison, we use the **public GQA benchmark** [24], which similarly demands both visual and textual understanding and is representative of such multi-image, visual-question-answering-style tasks. We evaluate RTP-LLM's Vision Transformer (ViT) decoupling (EPD) performance for the Qwen/Qwen2.5-VL-7B-Instruct model [9, 48, 51] on GQA, comparing against SGLang and vLLM. All runs use Tensor Parallelism (TP=2) deployment across two GPUs and consistent parameters: Max_batch_size = 64, max_new_tokens = 500, top-p = 1, top-k = 1, temperature = 0.0, ensuring fair comparison across different frameworks.

Figure 7 presents the performance and GPU memory utilization comparison across three frameworks. RTP-LLM demonstrates superior performance across all metrics. In terms of throughput (Figure 7(c)), RTP-LLM achieves 6288.48 tokens/s, which is 1.86x faster than SGLang (3374.24 tokens/s) and 2.52x faster than vLLM (2492.69 tokens/s). In terms of the latency metric, RTP-LLM achieves the lowest TTFT at 1737.48 ms (Figure 7(b)), representing 2.36x and 2.12x reduction compared to SGLang (4103.1 ms) and vLLM (3688.3 ms), respectively. RTP-LLM also achieves the lowest time cost at 3823.24 ms (Figure 7(a)), reducing latency by 20.1% and 21.5% compared to SGLang (4784 ms) and vLLM (4874 ms), respectively. This superior performance stems from RTP-LLM's decoupled architecture, which enables overlap between ViT processing and language model execution—particularly beneficial for production workloads with multi-image input and image-retrieval reference context, where vision encoding and text generation can be pipelined efficiently.

Figure 7(d) presents the GPU memory utilization comparison. RTP-LLM's EPD Disaggregation architecture enables efficient memory distribution across GPUs: GPU0 uses only 9,279.81 MB while GPU1 uses 89,087.81 MB, demonstrating significant memory savings on the first GPU. In contrast, SGLang and vLLM distribute memory more evenly across both GPUs under TP=2 deployment, with average memory usage of 82,210.34 MB (average of 82,442.12 MB and 81,978.56 MB across two GPUs) and 97,871 MB (average across two GPUs), respectively. This asymmetric memory distribution in RTP-LLM is achieved through the decoupled design that separates vision encoding (processed on GPU0 with minimal memory footprint) from language generation (processed on GPU1), resulting in better resource utilization and reduced computational overhead, as evidenced by the superior throughput and latency metrics.

9 Conclusions

We present RTP-LLM, a high-performance and production-ready inference engine designed to meet the rigorous demands of industrial-scale LLM deployment within Alibaba Group. RTP-LLM optimizes production-scale model loading to enable rapid elastic scaling, achieves superior memory efficiency via adaptive quantization and KV cache management, implements robust distributed processing using the Prefill-Decode Disaggregation architecture, and crucially, deploys effective production traffic scheduling strategies to maximize GPU utilization and guarantee Service Level Objectives (SLOs) under complex, high-concurrency workloads. Experimental results demonstrate that RTP-LLM delivers substantial advantages in throughput and latency compared to state-of-the-art open-source baselines, using both public benchmarks and real production workloads; the system successfully powers mission-critical applications across various Alibaba business units. RTP-LLM is released as open-source software, and the design choices, configurations. In future, we plan to further explore acceleration techniques such as the **DSA** (DeepSeek Sparse Attention) mechanism.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Amey Agrawal, Nitin Kedia, Anmol Agarwal, Jayashree Mohan, Nipun Kwatra, Souvik Kundu, Ramachandran Ramjee, and Alexey Tumanov. 2025. On Evaluating Performance of LLM Inference Serving Systems. *arXiv preprint arXiv:2507.09019* (2025).

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav 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 24)*. 117–134.

[4] Alibaba. 2025. Source code of RTP-LLM. https://github.com/alibaba/rtp-llm.

[5] Aone. 2025. Aone Copilot. Visual Studio Code Extension. https://marketplace.visualstudio.com/items?itemName=Aone.aone-copilot Accessed: 2025-11-15.

[6] Hicham Badri and Appu Shaji. 2023. Half-Quadratic Quantization of Large Machine Learning Models. https://mobiusml.github.io/hqq_blog/

[7] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[8] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. arXiv:2308.12966 [cs.CV] https://arxiv.org/abs/2308.12966

[9] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. *arXiv preprint arXiv:2308.12966* (2023).

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[11] NVIDIA Corporation. 2025. NVIDIA Collective Communications Library (NCCL) User Guide. Online Documentation. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html Accessed: 2025-11-15.

[12] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

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

[14] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[15] Yangshen Deng, Zhengxin You, Long Xiang, Qilong Li, Peiqi Yuan, Zhaoyang Hong, Yitao Zheng, Wanting Li, Runzhong Li, Haotian Liu, et al. 2025. AlayaDB: The Data Foundation for Efficient and Effective Long-context LLM Inference. In *Companion of the 2025 International Conference on Management of Data*. 364–377.

[16] Alexey Dosovitskiy. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[17] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints* (2024), arXiv–2407.

[18] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323* (2022).

[19] Shihong Gao, Xin Zhang, Yanyan Shen, and Lei Chen. 2025. Apt-Serve: Adaptive Request Scheduling on Hybrid Cache for Scalable LLM Inference Serving. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[20] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[21] Kostas Hatalis, Despina Christou, Joshua Myers, Steven Jones, Keith Lambert, Adam Amos-Binks, Zohreh Dannenhauer, and Dustin Dannenhauer. 2023. Memory matters: The need to improve long-term memory in llm-agents. In *Proceedings of the AAAI Symposium Series*, Vol. 2. 277–280.

[22] Yongjun He, Yao Lu, and Gustavo Alonso. 2024. Deferred continuous batching in resource-efficient large language model serving. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. 98–106.

[23] Kalle Hilsenbek. 2024. Breaking the Attention Bottleneck. *arXiv preprint arXiv:2406.10906* (2024).

[24] Drew A. Hudson and Christopher D. Manning. 2019. GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering. arXiv:1902.09506 [cs.CL] https://arxiv.org/abs/1902.09506

[25] Hugging Face Inc. 2024. Text Generation Inference (TGI): High-Performance Inference Engine for Large Language Models. https://huggingface.co/text-generation-inference.

[26] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. 2024. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems* (2024).

[27] Saehan Jo and Immanuel Trummer. 2025. SpareLLM: Automatically Selecting Task-Specific Minimum-Cost Large Language Models under Equivalence Constraint. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[28] Uday Kamath, Kevin Keenan, Garrett Somers, and Sarah Sorenson. 2024. LLMs in Production. In *Large Language Models: A Deep Dive: Bridging Theory and Practice*. Springer, 315–373.

[29] Mikhail V Koroteev. 2021. BERT: a review of applications in natural language processing and understanding. *arXiv preprint arXiv:2103.11943* (2021).

[30] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[31] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[32] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. 2024. Llm inference serving: Survey of recent advances and opportunities. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[33] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077* (2024).

[34] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems* 6 (2024), 87–100.

[35] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[36] Guangda Liu, Chengwei Li, Jieru Zhao, Chenqi Zhang, and Minyi Guo. 2025. Clusterkv: Manipulating llm kv cache in semantic space for recallable compression. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[37] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. *Advances in neural information processing systems* 36 (2023), 34892–34916.

[38] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. 2025. Muon is Scalable for LLM Training. arXiv:2502.16982 [cs.LG] https://arxiv.org/abs/2502.16982

[39] Mikasenghaas and Hugging Face Dataset Authors. 2024. Wikitext-2 Dataset Mirror. Hugging Face Dataset Card. https://hf-mirror.com/datasets/mikasenghaas/wikitext-2 Accessed: 2024-11.

[40] NVIDIA. 2023. TensorRT LLM. https://github.com/NVIDIA/TensorRT-LLM.

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

[42] Satya Naga Mallika Pothukuchi. 2025. LLMOps: A Comprehensive Guide to Deploying Large Language Models in Production. *IJSAT-International Journal on Science and Technology* 16, 1 (2025).

[43] Ruoyu Qin, Zheming Li, Weiran He, Jialei 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 25)*. 155–170.

[44] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[45] Gursimran Singh, Xinglu Wang, Yifan Hu, Timothy Yu, Linzi Xing, Wei Jiang, Zhefeng Wang, Xiaolong Bai, Yi Li, Ying Xiong, Yong Zhang, and Zhenan Fan. 2025. Efficiently Serving Large Multimodal Models Using EPD Disaggregation. arXiv:2501.05460 [cs.DC] https://arxiv.org/abs/2501.05460

[46] Benjamin Spector and Chris Re. 2023. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623* (2023).

[47] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/

[48] Qwen Team. 2025. Qwen2.5-VL. https://qwenlm.github.io/blog/qwen2.5-vl/

[49] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[51] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. 2024. Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. *arXiv preprint arXiv:2409.12191* (2024).

[52] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 233–248.

[53] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, et al. 2025. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 5831–5841.

[54] Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *arXiv preprint arXiv:2401.07851* (2024).

[55] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 Technical Report. *arXiv preprint arXiv:2407.10671* (2024).

[56] Yuqing Yang, Lei Jiao, and Yuedong Xu. 2024. A queueing theoretic perspective on low-latency llm inference with variable token length. In *2024 22nd International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*. IEEE, 273–280.

[57] Takeshi Yoshimura, Tatsuhiro Chiba, Manish Sethi, Daniel Waddington, and Swaminathan Sundararaman. 2025. Speeding up Model Loading with fastsafetensors. arXiv:2505.23072 [cs.DC] https://arxiv.org/abs/2505.23072

[58] Haoran You, Yichao Fu, Zheng Wang, Amir Yazdanbakhsh, and Yingyan Celine Lin. 2024. When linear attention meets autoregressive decoding: Towards more effective and efficient linearized large language models. *arXiv preprint arXiv:2406.07368* (2024).

[59] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[60] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, et al. 2025. JENGA: Effective memory management for serving LLM with heterogeneity. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 446–461.

[61] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2023), 34661–34710.

[62] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. 2025. DeepEP: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP.

[63] Pinxue Zhao, Hailin Zhang, Fangcheng Fu, Xiaonan Nie, Qibin Liu, Fang Yang, Yuanbo Peng, Dian Jiao, Shuaipeng Li, Jinbao Xue, et al. 2025. MEMO: Fine-grained Tensor Management For Ultra-long Context LLM Training. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–28.

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

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

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