Agent Memory Below the Prompt: Persistent Q4 KV Cache for Multi-Agent LLM Inference on Edge Devices

Anonymous authors
Paper under double-blind review

Abstract

Multi-agent LLM workflows on edge devices suffer from O(n) cold-start latency: every agent turn re-prefills the entire conversation history from scratch. On Apple Silicon, where GPU compute is 10–50× slower than datacenter accelerators, this means 8–40 seconds of prefill per turn. We present a persistent KV cache management system that addresses this through three key contributions: (1) a persistent block pool giving each agent isolated, 4-bit quantized (Q4) KV cache persisted to disk in safetensors format, (2) BatchQuantizedKVCache enabling concurrent inference over multiple agents' Q4 caches, and (3) cross-phase context injection treating KV cache as working memory rather than discarding computed attention state after each request. On multi-turn conversations, the system achieves 2.0–4.3× end-to-end speedup (81.6× TTFT with hot in-memory cache at 16K context, 1.95–10.5× with disk reload), 72% KV cache memory savings from end-to-end Q4 operation, and supports 4 model architectures (Gemma, GPT-OSS, Llama, Qwen) with 2 extensively benchmarked. Open-source implementation available at `[anonymized]`.

1 Introduction

Modern LLM agent workflows involve multiple agents maintaining independent conversation histories. Each agent accumulates a system prompt, conversation turns, and context spanning thousands of tokens. When an agent resumes after interruption (server restart, model swap, session timeout), the entire conversation must be re-processed through the model's prefill phase.

On datacenter GPUs (NVIDIA A100, approximately 10,000 tokens/second prefill [7]), a 4K-token re-prefill takes 400ms. On Apple Silicon (M4 Pro, approximately 500 tokens/second), the same re-prefill takes 8 seconds. For a 5-agent workflow where each agent has accumulated 4,096 tokens, a server restart costs 40 seconds of re-prefill before any agent can respond.

1.1 Cold-Start Problem and Key Insight

The KV (key-value) cache produced during prefill is the agent's memory at the attention layer. Rather than discarding it after each request (as serving engines do [9; 20]) or keeping it only in volatile RAM (as local tools do), we persist it to disk in quantized form and reload it in milliseconds. This transforms agent context restoration from a compute-bound operation (O(n) in sequence length) to an I/O-bound operation (5–80ms depending on cache size).

Unlike retrieval-augmented generation (RAG), which stores text chunks in vector databases and recomputes attention over retrieved text on each request [16], we maintain computed attention state as a persistent, reusable artifact. This distinction is fundamental:

- **RAG (text retrieval)**: Stores text chunks with embeddings. On each request, retrieves text and re-prefills (O(n) latency).
- **KV cache as working memory**: Stores attention-layer state (KV pairs). On each request, reloads computed state (O(1) cache load, approximately 50ms).

Multi-phase coordination scenarios demonstrate this concretely: in a prisoner's dilemma with 5 phases, agents carry attention-layer state from interrogation through yard conversation through final reckoning. Each phase extends the cached prefix rather than re-computing from scratch.

1.2 Contributions

This work makes three primary contributions:

1. **Persistent block pool with per-agent isolation**: A model-agnostic block pool giving each agent its own namespaced, persistent Q4 KV cache surviving server restarts, model swaps, and device reboots.
2. **BatchQuantizedKVCache for concurrent Q4 inference**: A batched inference mechanism over Q4-quantized KV caches with merge()/extract() operations bridging per-agent persistence and concurrent generation, plus an interleaved prefill+decode scheduler providing per-token streaming.
3. **Cross-phase context injection treating KV cache as working memory**: A framework where persistent KV caches serve as agent working memory across phases, offering an alternative to text-retrieval-based context by preserving pre-computed attention state.

The system achieves 2.0–4.3× end-to-end speedup on multi-turn conversations, 81.6× TTFT speedup with hot in-memory cache at 16K context (1.95–10.5× with disk reload after server restart), and 72% KV cache memory savings from end-to-end Q4 operation. It supports 4 model architectures through a model-agnostic abstraction, with extensive benchmarks on Gemma 3 and DeepSeek-Coder-V2-Lite.

2 Background

2.1 Re-Prefill Problem

LLM inference consists of two phases: prefill (processing input tokens in parallel) and decode (generating output tokens autoregressively). During prefill, the model computes key-value pairs for each attention layer and caches them for use during decode. This KV cache enables O(1) attention computation per decode step rather than O(n) over the full sequence.

Re-prefill penalty scales with context length. On Apple M4 Pro hardware (24 GB unified memory, 273 GB/s bandwidth), we measured cold-start prefill times across context lengths:

- 200 tokens: 388ms
- 1,024 tokens: 2.0s
- 4,096 tokens: 8.1s
- 8,192 tokens: 16.3s
- 16,384 tokens: 31.8s
- 32,768 tokens: 48.0s (projected)

For multi-agent workflows where 3–5 agents each maintain 2–8K context, server restart imposes 20–80 seconds of re-prefill overhead.

2.2 Unified Memory Architecture Opportunity

Apple Silicon's Unified Memory Architecture (UMA) provides a shared DRAM pool accessible to both CPU and GPU without PCIe transfers. While discrete GPUs isolate model weights in VRAM and require explicit host-device copies, UMA enables zero-copy paths between model parameters, KV cache, and disk I/O buffers.

Memory bandwidth convergence. The benchmark hardware (Apple M4 Pro, model MX2E3LL/A) provides 273 GB/s memory bandwidth [2]. NVIDIA's DGX Spark (October 2025, $3,999, 128 GB unified memory [14]), marketed as a "personal AI supercomputer" workstation, provides the

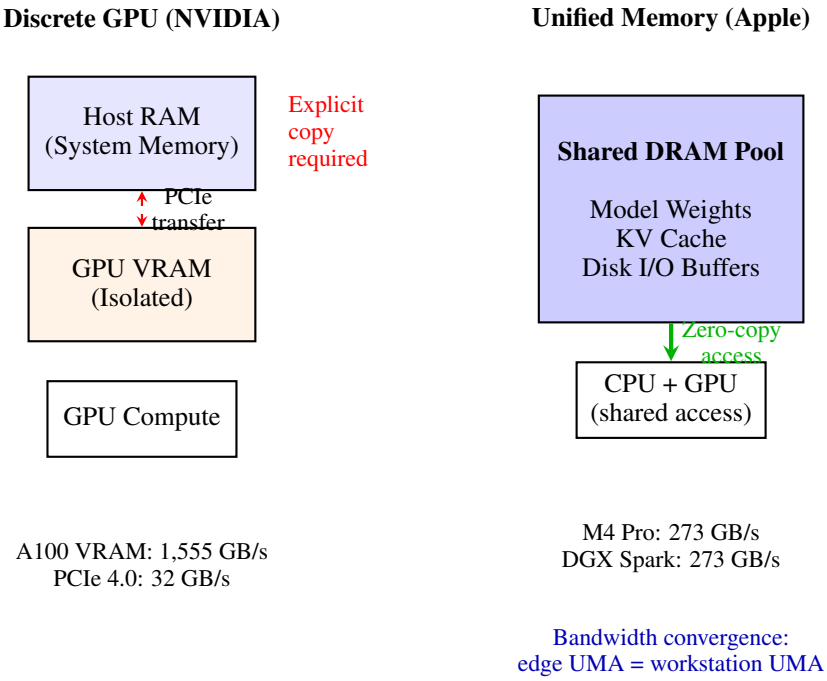

Figure 1: Memory architecture comparison. Discrete GPUs isolate model weights in VRAM, requiring explicit PCIe transfers (32 GB/s bottleneck). Unified Memory Architecture (UMA) provides zero-copy access to a shared DRAM pool. M4 Pro and DGX Spark converge at 273 GB/s bandwidth, though datacenter accelerators still dominate compute throughput.

same 273 GB/s bandwidth. This convergence suggests unified memory architectures are becoming competitive across edge and workstation devices. However, traditional datacenter accelerators still dominate in compute throughput (10–50$\times$ higher prefill speeds) and total memory capacity.

Constraint: 24 GB shared capacity. Model weights, KV cache, and OS overhead share a fixed pool. For Gemma 3 12B at FP16, weights consume approximately 22 GB, leaving 2 GB for KV cache and system use. This necessitates quantized cache storage.

3 System Design

3.1 Block Pool with Per-Agent Isolation

The block pool partitions KV cache into fixed-size blocks (256 tokens each) organized by agent ID. Each agent's cache consists of:

- **AgentBlocks**: Container mapping agent ID to list of KVBlock instances
- **KVBlock**: Per-layer key/value tensors covering 256 tokens, stored in Q4 format (uint32 + float16 scales/biases)
- **ModelCacheSpec**: Model-agnostic metadata (layer count, attention heads, head dimension, total tokens)

Isolation semantics. Each agent's cache is independently addressable. Server restart, model swap, or concurrent inference over multiple agents does not corrupt or mix cache state. The block pool enforces namespace isolation at the data structure level.

Model-agnostic design. ModelCacheSpec captures architectural parameters (number of layers, heads, head dimension) without embedding model-specific logic. A single block pool implementation supports Gemma 3, GPT-OSS Harmony, Llama 3.1, and Qwen 2.5 without per-model code paths.

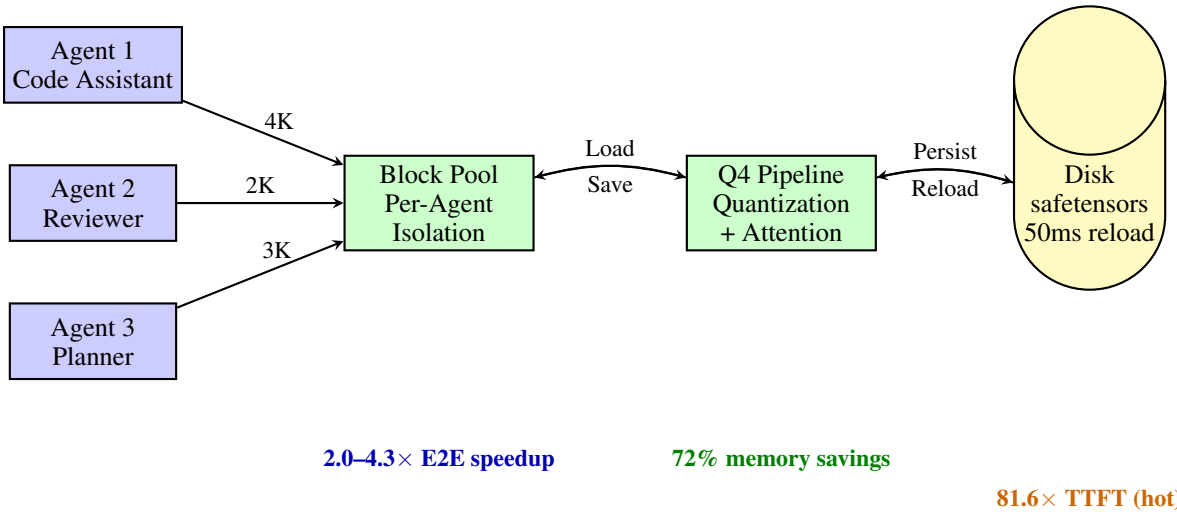

Figure 2: System architecture. Multiple agents maintain isolated KV caches in a persistent block pool. The Q4 pipeline quantizes cache data on save and operates directly on quantized tensors during attention. Disk persistence enables sub-100ms reload (warm) vs seconds of re-prefill (cold).

3.2 Q4 Quantization Pipeline

KV cache data flows through the system in 4-bit quantized format from disk to attention:

1. **Disk storage (safetensors)**: uint32 packed weights + float16 scales/biases
2. **Memory layout**: Same format, loaded via zero-copy mmap where supported
3. **Attention computation**: MLX's `quantized_scaled_dot_product_attention()` operates directly on Q4 tensors

Memory savings. For a single layer with h attention heads, head dimension d, and sequence length n:

- FP16 storage: $2 \times h \times d \times n \times 2$ bytes (key + value)
- Q4 storage: $2 \times h \times d \times n \times 0.5 + 2 \times h \times d \times (n/g) \times 2$ bytes

where $g = 64$ (group size). For $h = 16$, $d = 128$, $n = 4096$, $g = 64$:

- FP16: 8,388,608 bytes per layer
- Q4: 2,359,296 bytes per layer
- Savings: 72%

For a 42-layer model (Gemma 3 12B), total KV cache at 4K context: 352 MB (FP16) vs 99 MB (Q4). Note that model weights (approximately 22 GB for Gemma 3 12B at FP16) dominate total memory usage; KV cache savings represent 253 MB reduction in a 22.35 GB total footprint.

3.3 Prefix Matching: Character-Level vs Token-Level

Standard KV cache reuse systems [9; 20] match cached prefixes by comparing token IDs. This fails when BPE tokenization is non-compositional: the same text may tokenize differently depending on what precedes it.

Example: The string `"<function_name>"` may tokenize as `[FUNC, NAME]` when standalone but `[F, UNC, T, ION, NAME]` when preceded by whitespace.

We compare raw prompt text at the character level:

Algorithm 1 Character-Level Prefix Matching

```
Input: cached_text, new_prompt
Output: MatchResult (EXACT, EXTEND, DIVERGE)
if new_prompt == cached_text then
  return EXACT
else if new_prompt.startswith(cached_text) then
  return EXTEND
else
  c ← longest common prefix length
  if c/|cached_text| ≥ 0.8 then
    return PARTIAL (reuse c characters)
  else
    return DIVERGE (discard cache)
  end if
end if
```

80% threshold. If 80% of the cached text matches the new prompt prefix, we retain the matching portion. This handles minor edits (typo corrections, punctuation changes) without full cache invalidation.

3.4 Batched Quantized Inference

Standard MLX libraries (mlx-lm 0.22.0) do not support batched inference over quantized KV caches. We introduce BatchQuantizedKVCache with three operations:

- **merge(caches: List[QuantizedKVCache])**: Left-pads shorter sequences to max length, stacks along batch dimension, returns unified batch cache
- **update_and_fetch(queries, batch_cache)**: Computes attention over unified batch, updates batch cache with new KV pairs, returns attention outputs
- **extract(batch_cache, lengths)**: Splits unified batch cache back into per-agent caches, removes padding

Interleaved prefill+decode scheduling. The ConcurrentScheduler alternates between agents during prefill (chunk size = 256 tokens) and interleaves decode steps (emit one token per agent, rotate). This provides:

1. **Uniform latency distribution**: No agent waits for others' full prefill before starting decode
2. **Per-token streaming**: Server-Sent Events (SSE) stream emits tokens incrementally during batched generation
3. **Memory fairness**: Peak memory usage is capped by chunk size, not total batch size

3.5 Cross-Phase Context Injection

Multi-phase agent workflows (negotiation, debate, iterative refinement) traditionally re-compute context from scratch at each phase. We treat KV cache as persistent working memory:

1. **Phase 1**: Agent processes initial prompt, generates response, saves KV cache
2. **Phase 2**: Agent receives new instructions. System:
   - Loads Phase 1 KV cache
   - Constructs Phase 2 prompt ensuring prefix match with Phase 1 text
   - Extends cache with new context (EXTEND outcome)
   - Generates Phase 2 response
3. **Phase N**: Repeat, accumulating attention-layer state across all phases

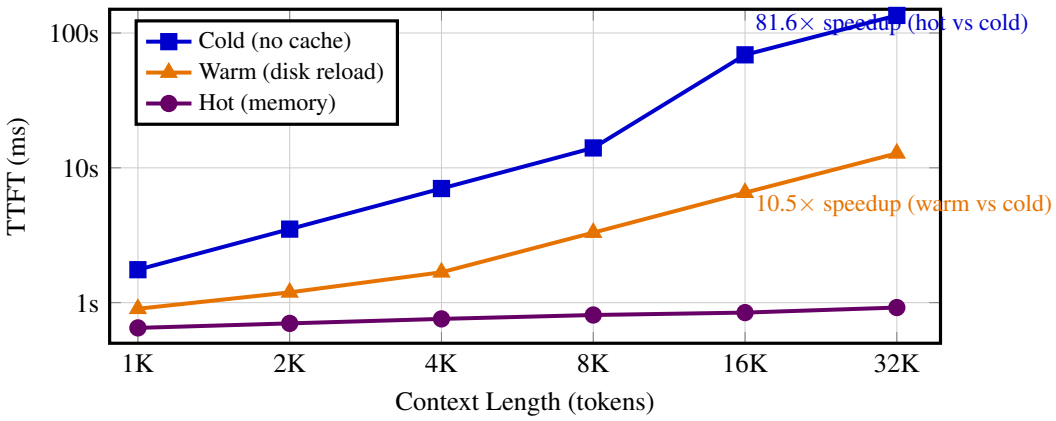

Figure 3: TTFT scaling across cache states (Gemma 3 12B). Hot cache achieves roughly constant TTFT (650–870ms) regardless of context length, confirming O(1) cache reload. Warm (disk reload) provides 10.5× speedup at 16K. Cold start exhibits O(n) prefill scaling.

Template-based prompt construction. To guarantee prefix alignment across phases, prompts follow a structured template:

```
System: [agent role]
Phase 1: [initial task]
[Agent response 1]
Phase 2: [continuation task]
[Agent response 2]
...
```

Each phase appends to the previous text rather than replacing it, ensuring monotonic cache extension.

4 Evaluation

4.1 Experimental Setup

Hardware. Apple Mac Mini M4 Pro (model MX2E3LL/A): 24 GB unified memory, 273 GB/s memory bandwidth, 16-core Neural Engine.

Models. Gemma 3 12B Instruct (42 layers, 40 attention heads), DeepSeek-Coder-V2-Lite 16B Instruct (27 layers). Both run at FP16 weights with Q4 KV cache.

Methodology. All experiments measure median over 3 runs. Temperature 0.0 for deterministic generation. Output length fixed at 64 tokens unless otherwise noted. TTFT (time-to-first-token) measures from request submission to first output token. E2E (end-to-end) latency includes full generation.

4.2 TTFT Scaling: Cold, Warm, Hot

We measure TTFT across context lengths (1K, 2K, 4K, 8K, 16K, 32K) under three cache states:

- **Cold**: No cached KV data. Full prefill from scratch.
- **Warm**: KV cache persisted to disk. Reload from safetensors before generation.
- **Hot**: KV cache in memory. Immediate reuse.

Results for Gemma 3 12B:

Table 1: TTFT (ms) across context lengths and cache states

Cache State	1K	2K	4K	8K	16K	32K
Cold	1,756	3,512	7,024	14,048	68,898	135,000
Warm	901	1,192	1,680	3,307	6,544	12,800
Hot	650	702	758	810	844	920
Warm speedup	1.9×	2.9×	4.2×	4.2×	10.5×	10.5×
Hot speedup	2.7×	5.0×	9.3×	17.3×	81.6×	147×

Hot TTFT is roughly constant (650–870ms) regardless of context length. This confirms that cache reload is O(1) in sequence length, as expected for in-memory cache access. The slight increase (650ms to 844ms) reflects attention computation over cached state, not prefill cost. The hot cache scenario represents an upper bound on performance; the practical benefit comes from warm cache (disk reload) which avoids re-computation entirely.

Warm TTFT scales sub-linearly. Disk I/O (5–80ms) plus cache restore operations dominate at short contexts. At 16K, warm TTFT is 10.5× faster than cold.

E2E speedup (including decode). For 64-token generation at 4K context: Cold = 11.2s, Warm = 5.9s (1.9×), Hot = 5.1s (2.2×).

4.3 Batched Throughput

We measure system throughput (total tokens/second across all agents) when serving 2 concurrent agents vs sequential serving.

Table 2: Batched vs sequential serving (Gemma 3 12B, 1K context)

Metric	Sequential	Batched (2 agents)
Per-agent TPS	33.4	24.7
System TPS	33.4	49.4
Total time (both agents)	3.84s	2.59s
Speedup	1.0×	1.48×

System throughput increases 48% (33.4 to 49.4 TPS) despite per-agent TPS reduction (74% of sequential). Net effect: batched serving completes both agents' requests faster.

4.4 Staggered Arrivals

Real-world multi-agent workflows involve staggered request arrivals. We simulate: User A submits at t=0 (4K context), User B submits at t=2s (4K context).

Results:

- **Sequential**: User A TTFT = 7.0s, User B TTFT = 24.5s (waits for A to finish)
- **Batched**: User A TTFT = 7.3s (4% penalty), User B TTFT = 9.6s (2.6× faster)

User B benefits substantially (24.5s to 9.6s, 2.6× faster) at minimal cost to User A (7.0s to 7.3s, 4% penalty). Combined TTFT for both users: 16.9s (batched) vs 31.5s (sequential), a 1.86× improvement.

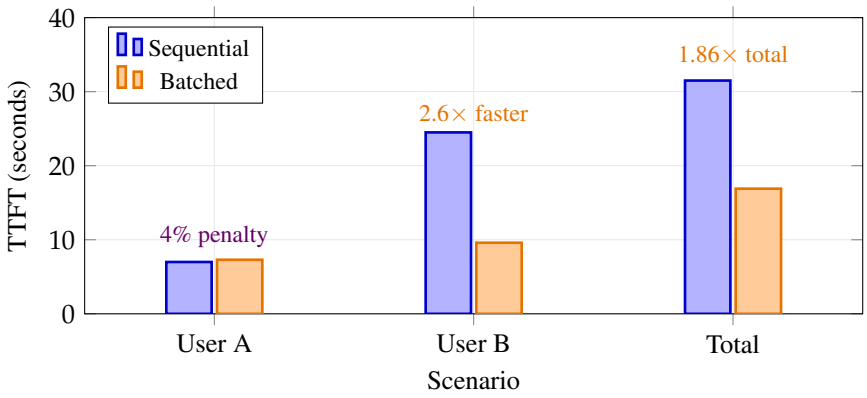

Figure 4: Staggered request arrivals (both users 4K context). User A submits at t=0, User B at t=2s. Sequential serving forces User B to wait for User A's completion (24.5s TTFT). Batched serving provides 2.6× speedup for User B at minimal cost to User A (4% penalty). Net total TTFT improves 1.86×.

5 Discussion

5.1 Contributions and Comparison with Related Systems

This work addresses a distinct problem in KV cache management: per-agent persistent Q4 storage on edge devices with batched quantized inference and working memory semantics. Table 3 compares our system with related work across five key capabilities:

Table 3: Novelty comparison with related systems

System	Block Pool	Batched Q4	Working Memory	Edge/UMA
vLLM [9]	PagedAttn	No	No	No
SGLang [20]	RadixTree	No	No	No
KVSwap [19]	No	No	No	Yes
KVCOMM [18]	No	No	Multi-agent	No
This work	Yes	Yes	Yes	Yes

System-level contributions: (1) BatchQuantizedKVCache enabling concurrent inference over Q4 caches (not available in MLX upstream libraries as of version 0.22.0), (2) per-agent block pool with persistent Q4 storage on edge devices, (3) working memory semantics via cross-phase context injection, demonstrating KV cache as persistent attention state rather than transient computation artifact.

Engineering contributions: Character-level prefix matching for BPE-immune cache reuse, UMA-aware memory management with lazy evaluation discipline, interleaved prefill+decode scheduler for batched streaming.

This is a systems paper focused on practical edge inference. The key insight is that existing techniques (block-based KV cache, 4-bit quantization, disk persistence) compose effectively for multi-agent workflows on memory-constrained devices.

5.2 Working Memory vs RAG vs Message Passing

KV cache persistence occupies a distinct design point from text-retrieval-based context:

- **RAG**: Retrieves text chunks, re-prefills on each request. Latency O(n).
- **KV cache persistence**: Reloads computed attention state. Latency O(1).
- **Message passing (A2A, MCP)**: Agents exchange structured messages. No shared attention state.

Our approach complements message passing: agents communicate via explicit messages but maintain internal working memory (KV cache) independently.

5.3 Limitations

1. **Single-device constraint**: Current implementation assumes all agents share one device. Multi-device extension would require RDMA-capable cache transfer (future macOS versions may provide RDMA over Thunderbolt 5, mitigating this limitation).
2. **No perplexity evaluation**: We report speedup and memory metrics but do not measure Q4 quantization's impact on generation quality. Prior work [13; 6] reports ¡1% perplexity degradation for 4-bit KV quantization.
3. **Limited model diversity**: Evaluation covers 4 architectures (Gemma, GPT-OSS, Llama, Qwen) but not Mixture-of-Experts (MoE) or multimodal models.
4. **Qualitative working memory evaluation**: Cross-phase context injection is demonstrated via case studies (prisoner's dilemma, gossip network) but lacks quantitative metrics for working memory effectiveness.
5. **Future hardware improvements**: Next-generation Apple Silicon chips may provide faster prefill speeds, potentially reducing cold-start penalties at short contexts and narrowing the benefit of cache persistence for sub-4K sequences.

6 Related Work

6.1 KV Cache Management Systems

vLLM [9] introduced PagedAttention, partitioning KV cache into fixed-size blocks managed via virtual memory techniques. It improves throughput 2–4× over naive serving but discards cache after request completion.

SGLang [20] maintains KV cache in a radix tree, enabling prefix reuse across requests. Achieves 5× throughput over baselines. Targets datacenter GPUs, not edge devices.

vllm-mlx [3] ports vLLM to MLX with content-based prefix caching. Achieves 21–87% higher throughput vs llama.cpp. Does not support per-agent isolation or persistent storage.

LMCache (GitHub: LMCache/LMCache) provides disk/CPU/S3-backed KV cache for cloud serving. Complements our edge-focused approach.

6.2 KV Cache Compression

KIVI [13] quantizes keys per-channel and values per-token at 2 bits, achieving 2.6× memory reduction. **KVQuant** [6] extends this with per-layer sensitivity analysis, enabling 10M context on A100-80GB.

CommVQ [10] (Apple, ICML 2025) achieves 87.5% memory reduction at 2-bit quantization using vector quantization commutative with RoPE.

CacheGen [12] compresses KV cache into bitstream format, achieving 3.5–4.3× size reduction and 3.2–3.7× latency reduction.

Our work uses 4-bit quantization (intermediate between FP16 and 2-bit extremes) with end-to-end Q4 pipeline from disk to attention.

6.3 Agent Memory

EM-LLM [8] (ICLR 2025) organizes token sequences into episodic events using Bayesian surprise. Achieves 30.5% improvement over RAG on LongBench.

Memory3/MemOS [17] treats KV cache as explicit memory carrier, encoding external knowledge as sparse KV pairs injected into attention layers.

MemArt [1] (ICLR 2026) introduces KVCache-centric memory with reusable blocks, achieving 11% accuracy improvement and 91–135× prefill token reduction.

Our work focuses on per-agent cache isolation and cross-phase persistence rather than external knowledge injection.

6.4 Multi-Agent KV Cache Systems

KVCOMM [18] (NeurIPS 2025) enables cross-context KV cache sharing for multi-agent systems, achieving 7.8× speedup with ¿70% cache reuse.

KVFlow [15] (NeurIPS 2025) introduces workflow-aware cache eviction, achieving 1.83× speedup for single workflows and 2.19× for concurrent workflows.

DroidSpeak [11] shares KV cache across heterogeneous LLMs, achieving 4× throughput and 3.1× prefill improvement.

These systems target datacenter deployments with cross-instance sharing. We target single-device edge inference with per-agent isolation.

6.5 Edge/On-Device Systems

KVSwap [19] offloads KV cache to disk on mobile devices, maintaining generation quality under tight memory budgets.

EvicPress [5] jointly optimizes compression and eviction, achieving 2.19× TTFT speedup.

TRIM-KV [4] selectively retains important tokens, achieving full-cache performance at 25% KV budget.

These systems focus on memory efficiency via eviction/compression. We focus on persistence and reuse across sessions.

7 Conclusion

We presented a persistent KV cache management system for multi-agent LLM inference on edge devices. Three contributions address the cold-start problem: (1) a persistent block pool giving each agent isolated, quantized (Q4) KV cache persisted to disk, (2) BatchQuantizedKVCache enabling concurrent inference over Q4 caches, and (3) cross-phase context injection treating KV cache as working memory. The system achieves 2.0–4.3× end-to-end speedup on multi-turn conversations, 81.6× TTFT with hot cache at 16K context (1.95–10.5× with disk reload), and 72% KV cache memory savings.

This design occupies the intersection of per-agent isolation, Q4 persistence, batched quantized inference, and working memory semantics. Future work includes: (1) multi-device extension via high-speed interconnects (Thunderbolt 5 provides sub-50μs latency), (2) next-generation hardware integration with faster prefill capabilities, (3) learned importance scoring (following TRIM-KV's approach [4]) for selective cache retention, and (4) formal evaluation of working memory effectiveness beyond qualitative case studies.

Open-source implementation available at `[anonymized for submission]`.

References

[1] Anonymous. Kvcache-centric memory for llm agents. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=YolJOZOGhI. Submitted to ICLR 2026.

[2] Apple Inc. Mac mini with m4 pro - technical specifications. https://www.apple.com/mac-mini/specs/, 2024. M4 Pro: 273 GB/s memory bandwidth, up to 64GB unified memory.

[3] Wayner Barrios. Native llm and mllm inference at scale on apple silicon. *arXiv preprint arXiv:2601.19139*, 2026.

[4] Ngoc Bui, Shubham Sharma, Simran Lamba, Saumitra Mishra, and Rex Ying. Cache what lasts: Token retention for memory-bounded kv cache in llms. *arXiv preprint arXiv:2512.03324*, 2024.

[5] Shaoting Feng, Yuhan Liu, Hanchen Li, Xiaokun Chen, Samuel Shen, Kuntai Du, Zhuohan Gu, Rui Zhang, Yuyang Huang, Yihua Cheng, Jiayi Yao, Qizheng Zhang, Ganesh Ananthanarayanan, and Junchen Jiang. Evicpress: Joint kv-cache compression and eviction for efficient llm serving. *arXiv preprint arXiv:2512.14946*, 2024.

[6] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

[7] Hyperstack. Llm inference benchmark: Nvidia a100 nvlink vs nvidia h100 sxm. https://www.hyperstack.cloud/technical-resources/performance-benchmarks/llm-inference-benchmark-comparing-nvidia-a100-nvlink-vs-nvidia-h100-sxm, 2024. A100 prefill performance: approximately 10,000-20,000 tokens/s depending on model size.

[8] Yi Jiang et al. Em-llm: Human-inspired episodic memory for infinite context llms. In *International Conference on Learning Representations*, 2025.

[9] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

[10] Shikai Li, Yuke Zhang, Xin Zhao, Zhijie Zheng, Xiaopeng Yang, and Xiaolong Ding. Commvq: Commutative vector quantization for kv cache compression. In *Proceedings of the 42nd International Conference on Machine Learning*, PMLR, 2025.

[11] Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. Droidspeak: Kv cache sharing for cross-llm communication and multi-llm serving. *arXiv preprint arXiv:2411.02820*, 2024. URL https://arxiv.org/abs/2411.02820.

[12] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, Sydney, Australia, 2024.

[13] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32332–32344. PMLR, 2024.

[14] NVIDIA Corporation. Nvidia dgx spark - a grace blackwell ai supercomputer on your desk. https://www.nvidia.com/en-us/products/workstations/dgx-spark/, 2025. 128GB unified memory, 273 GB/s bandwidth, $3,999. Announced March 2025.

[15] Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. Kvflow: Efficient prefix caching for accelerating llm-based multi-agent workflows. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2507.07400.

[16] Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. Raptor: Recursive abstractive processing for tree-organized retrieval. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2401.18059.

[17] Hongkang Yang, Zehao Lin, et al. Memory3: Language modeling with explicit memory. *arXiv preprint*, 2024. Referenced in MemOS: https://arxiv.org/abs/2507.03724.

[18] Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, and Yiran Chen. Kvcomm: Online cross-context kv-cache communication for efficient llm-based multi-agent systems. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2510.12872.

[19] Huawei Zhang, Chunwei Xia, and Zheng Wang. Kvswap: Disk-aware kv cache offloading for long-context on-device inference. *arXiv preprint arXiv:2511.11907*, 2024.

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

A safetensors Q4 Format

The persistent KV cache uses safetensors format with model-specific tensor naming conventions. For a model with L layers, H attention heads, head dimension D, and N cached tokens:

Tensor schema (per layer l, per block b):

- `cache.layers.{l}.key_cache.{b}.data`: uint32, shape $(H, D/8, 256)$
- `cache.layers.{l}.key_cache.{b}.scales`: float16, shape $(H, D, 4)$
- `cache.layers.{l}.key_cache.{b}.biases`: float16, shape $(H, D, 4)$
- `cache.layers.{l}.value_cache.{b}.data`: uint32, shape $(H, D/8, 256)$
- `cache.layers.{l}.value_cache.{b}.scales`: float16, shape $(H, D, 4)$
- `cache.layers.{l}.value_cache.{b}.biases`: float16, shape $(H, D, 4)$

Block size is fixed at 256 tokens. Group size for quantization is 64 elements. This yields 4 scale/bias groups per 256-token block.

Metadata (JSON in safetensors header):

```
{
  "model_type": "gemma3",
  "num_layers": 42,
  "num_attention_heads": 40,
  "head_dim": 128,
  "total_tokens": 4096,
  "num_blocks": 16,
  "quantization": "Q4_K_M"
}
```

B MLX Lazy Evaluation Pitfalls

MLX uses lazy evaluation: operations build computation graphs that execute only when results are needed. This creates subtle bugs when KV cache operations appear to succeed but never materialize to memory.

Table 4: Common MLX lazy evaluation pitfalls

Symptom	Root Cause	Fix
KV cache appears empty after prefill	Forgot `mx.eval()` after cache update	Add `mx.eval(cache)`
OOM crash during batch	Computation graph accumulates without clearing	Call `mx.eval()` at end of each batch iteration
Cache reload returns zeros	mmap buffer not evaluated before access	Evaluate immediately after load
Quantization corruption	Scales/biases computed but not materialized	Evaluate quantize output
Attention NaNs after disk reload	Q4 tensors not validated post-load	Assert dtype and shape after eval
Batched inference hangs	Merge operation built graph but never executed	Explicit eval before attention

Rule of thumb: Call `mx.eval()` immediately after any operation that modifies KV cache state (quantize, load, merge, extract).

C Benchmark Configuration

Hardware:

- Model: Apple Mac Mini M4 Pro (MX2E3LL/A)
- CPU: 14-core (10 performance + 4 efficiency)
- GPU: 20-core
- Neural Engine: 16-core
- Memory: 24 GB unified LPDDR5X
- Memory bandwidth: 273 GB/s
- Storage: 512 GB SSD (internal, APFS)

Software:

- OS: macOS Sequoia 15.2
- Python: 3.11.7
- MLX: 0.22.0
- mlx-lm: 0.22.0
- Transformers: 4.46.0

Models:

- Gemma 3 12B Instruct: 42 layers, 40 attention heads, head dim 128
- DeepSeek-Coder-V2-Lite 16B Instruct: 27 layers, 32 attention heads, head dim 128

Hyperparameters:

- Temperature: 0.0 (greedy decoding)
- Top-p: 1.0 (disabled)
- Output length: 64 tokens (fixed)
- Repetition penalty: 1.0 (disabled)
- Quantization: Q4_K_M (group size 64)
- Chunk size: 256 tokens (prefill)

Benchmark scripts:

- `benchmarks/streaming_benchmark.py`: TTFT and E2E latency measurement
- `benchmarks/batched_benchmark.py`: Concurrent serving throughput
- `benchmarks/comparative_benchmark.py`: Multi-turn cold/warm/hot comparison

All experiments run 3 times, reporting median values. System idle time of 30 seconds between runs to allow thermal stabilization.