Comparative Analysis: Stateful Cache-Aware Routing vs. SOTA

Project Team

November 23, 2025

Executive Summary

This report analyzes the current State-of-the-Art (SOTA) in distributed LLM inference caching—specifically **vLLM**, **LMcache**, **llm-d**, and **SGLang**—and compares them with our proposed **Stateful Cache-Aware Router with Eviction Callbacks**.

Conclusion: Our project represents a valid and distinct contribution to the field. While existing solutions focus on *local optimization* (vLLM/SGLang) or *data movement* (LMcache), our approach focuses on **lightweight, consistency-aware routing**. The introduction of "Push-based Eviction Callbacks" addresses the "Stale Cache" problem in distributed routing without the overhead of distributed storage or heavy orchestration.

1 State-of-the-Art (SOTA) Analysis

1.1 vLLM: The Local Optimization Champion

- **Mechanism**: "Automatic Prefix Caching" (APC) using hash-based block matching.
- **Strengths**: Extremely fast for *local* cache hits. Zero network overhead.
- **Limitations**: **Cache Blindness**. In a distributed cluster, Node A does not know what Node B has cached. A standard load balancer might route a request to Node B even if Node A has the hot cache, leading to redundant prefill.
- **Gap**: Lacks a native distributed routing layer.

1.2 LMcache: The Storage Disaggregation Approach

- **Mechanism**: Decouples KV cache from compute. Offloads cache to CPU/Disk/Remote (Redis).
- **Strengths**: Infinite capacity (theoretically). Enables context sharing across sessions.
- **Limitations**: **Latency**. Fetching KV cache from remote storage (even over RDMA) incurs network serialization/deserialization costs. For short-to-medium prompts, recomputing might be faster than fetching.
- **Gap**: Focuses on *moving data to compute*, whereas we focus on *moving compute to data*.

1.3 llm-d: The Kubernetes Orchestrator

- **Mechanism**: K8s-native scheduler with an "Inference Gateway". Tracks pod state to route requests.
- **Strengths**: Production-ready, integrates with K8s autoscaling.
- **Limitations**: **Complexity & Granularity**. Heavily coupled with Kubernetes. Often relies on polling or coarse-grained metrics.
- **Gap**: High barrier to entry. May not handle real-time cache eviction events with the same granularity as a direct engine hook.

1.4 SGLang: The Radix Tree Innovator

- **Mechanism**: RadixAttention (Trie-based) instead of hash blocks.
- **Strengths**: Handles partial matches and complex branching better than vLLM.
- **Limitations**: Like vLLM, it is primarily a single-node engine optimization. Distributed SGLang relies on data parallelism but faces the same routing consistency challenges.

2 Our Contribution: Stateful Router with Eviction Callbacks

Our system sits as a lightweight middleware layer that bridges the gap between the **Engine** (vLLM) and the **Cluster** (Router).

2.1 The Core Innovation: "Push-Based Consistency"

Most routers use "Pull" (polling load) or "Estimation" (assuming cache exists). We introduce a **Push** mechanism:

1. **Inference**: Router speculatively updates the map (`Hash` $\rightarrow$ `Worker`).
2. **Eviction**: Worker *actively notifies* the router when a block is freed.

2.2 Comparative Matrix

Feature	vLLM	LMcache	llm-d	Our Solution
Routing Strategy	Round-Robin	N/A (Storage)	K8s Gateway	**Sticky-Least-Loaded**
Data Locality	None	Low	High	**High (Route-to-Data)**
State Consistency	N/A	Strong	Eventual	**Eventual (Push)**
Network Overhead	Low	High	Medium	**Very Low**
Implementation	Engine C++	Middleware	K8s Operator	**Python Middleware**

2.3 Why This is Valid & Useful

1. **Zero-Copy Efficiency**: Unlike LMcache, we never move the heavy KV cache over the network. We simply route the tiny request to where the cache already lives.
2. **Granular Consistency**: By hooking into `free_block`, we get exact visibility into cache lifecycle. If a worker is under memory pressure and evicts a prefix, the router knows immediately and stops sending requests there, preventing "False Hits".
3. **Agnostic Design**: Our router is independent of Kubernetes (unlike llm-d) and storage backends (unlike LMcache). It can run on bare metal, Slurm, or Docker Compose.

3 Conclusion

The **Stateful Cache-Aware Router** is not just a reimplementation of existing tools. It occupies a specific niche: **optimizing distributed cache hit rates via lightweight coordination**.

It solves the "Distributed Cache Blindness" problem of vLLM without incurring the "Data Movement Latency" of LMcache or the "Infrastructure Complexity" of llm-d. This makes it a highly relevant project for research into efficient LLM serving systems.

A Deep Dive: Technical Implementation Details

This appendix provides a granular analysis of the implementation strategies for the discussed systems, pinpointing specific files and algorithms where possible.

A.1 vLLM: Automatic Prefix Caching (APC)

Strategy: Hash-based block reuse within a single engine instance.

- **Key File**: `vllm/core/block/prefix_caching_block.py` (and `block_manager.py`)
- **Prefix Caching Algorithm**:
  1. **Hashing**: As tokens are processed, vLLM computes a hash of the token sequence in the current block + the hash of the prefix block.
  2. **BlockTracker**: Maintains a mapping of `BlockHash` $\rightarrow$ `PhysicalBlock`.
  3. **Allocation**: When allocating a new block for a sequence, the `BlockAllocator` checks if a block with the computed hash already exists in the `FreeList` or `CachedList`.
  4. **Eviction**: Uses an LRU policy. When memory is full, blocks with a reference count of 0 (not currently used by any active sequence) are evicted from the `CachedList`.
- **Limitation**: The `BlockTracker` is local to the `LLMEngine` instance. There is no mechanism to broadcast the existence of a cached block to other workers or a global router.

A.2 LMcache: Multi-Tier KV Cache Offloading

Strategy: Decouple KV cache storage from the compute engine to allow persistence and sharing.

- **Key Files**:
  - `lmcache/storage_backend/abstract_backend.py`: Defines the interface for storage backends.
  - `lmcache/storage_backend/local_backend.py`: Implementation for local disk/CPU.
  - `lmcache/storage_backend/redis_backend.py`: Implementation for remote Redis storage.
- **KV Cache Routing/Storage**:
  1. **Connector**: A hook inside the inference engine (e.g., vLLM) intercepts KV cache operations.
  2. **Write (Offload)**: Asynchronously writes evicted or completed KV blocks to the configured backend (Redis/Disk).
  3. **Read (Prefetch)**: On a cache miss in GPU memory, it queries the backend. If found, it fetches the KV data, deserializes it, and moves it to GPU memory.
- **Latency**: The critical path involves `GPU` $\leftrightarrow$ `CPU` $\leftrightarrow$ `Network` $\leftrightarrow$ `Remote Storage`. Even with RDMA, this serialization latency often exceeds the compute time for short prefixes.

A.3 llm-d: Kubernetes-Native Cache-Aware Scheduling

Strategy: Intelligent request routing via a K8s Gateway that tracks pod load and cache state.

- **Key Component**: `Endpoint Picker (EPP)` in `llm-d-inference-scheduler`.
- **Routing Algorithm**:
  1. **Discovery**: Uses K8s API to discover `vLLM` pods.
  2. **State Tracking**: Maintains an in-memory map of which prefixes (or LoRA adapters) are active on which pods. This is often updated via piggybacked metrics or polling.
  3. **Scoring**: Incoming requests are scored against available pods.
  $$Score(pod) = w_1 \cdot CacheHit + w_2 \cdot (1 - Load)$$
  4. **Dispatch**: The request is routed to the pod with the highest score.
- **Complexity**: Requires deploying a custom K8s Controller, Gateway API, and specialized sidecars.

A.4 SGLang: RadixAttention

Strategy: Tree-based cache management instead of hash-based blocks.

- **Key File**: `python/sglang/srt/managers/radix_attention.py`
- **Algorithm**:
  1. **Radix Tree**: Maintains a CPU-side Radix Tree where edges are token sequences and nodes map to GPU memory indices.
  2. **Prefix Matching**: Unlike vLLM's block hash (which requires exact block alignment), SGLang can match partial prefixes of arbitrary length by traversing the tree.
  3. **Eviction**: LRU eviction is performed on leaf nodes of the Radix Tree.
- **Benefit**: Higher hit rate for dynamic system prompts or multi-turn conversations where the shared prefix might not align perfectly with block boundaries.

A.5 Sarathi-Serve: Chunked Prefills Stall-Free Scheduling

Strategy: Optimize throughput-latency trade-off by splitting prefill phases.

- **Key Concept**: **Chunked Prefills**.
- **Algorithm**:
  1. **Split**: A long prefill request is split into smaller chunks (e.g., 512 tokens).
  2. **Schedule**: The scheduler creates "Hybrid Batches" containing a mix of:
     - Decode tokens from ongoing requests.
     - One or more prefill chunks from new requests.
  3. **Stall-Free**: By limiting the size of the prefill chunk, the system ensures that the "Time Between Tokens" (TBT) for the decode requests remains low, avoiding the "head-of-line blocking" problem typical of FCFS schedulers.
- **Relevance**: While Sarathi focuses on *single-node scheduling efficiency*, our project focuses on *multi-node routing efficiency*. The two are complementary.