Adaptive Load-Balanced Routing for Distributed LLM Inference: Token-Aware SLO Gating with Escape Heuristics

Benjamin Li
Computer Science and Engineering
Michigan State University
East Lansing, MI
libinghe@msu.edu

Prabhaav Pillai
Computer Science and Engineering
Michigan State University
East Lansing, MI
prabhaav@msu.edu

Michael Tan
Computer Science and Engineering
Michigan State University
East Lansing, MI
tanmicha@msu.edu

***Abstract*—Distributed Large Language Models (LLMs) inference clusters face a fundamental routing tension: routing requests to replicas that already hold matching Key-Value (KV) cache entries minimizes redundant prefill computation and reduces Time To First Token (TTFT), yet concentrating traffic on popular-prefix replicas creates hotspots that inflate tail latency and reduce system-wide throughput. Existing systems address this tension incompletely. Cache-affinity routing causes load skew; load-only dispatching wastes GPU cycles through cache fragmentation; and the DualMap baseline, while combining both signals with a power-of-two choices ($d = 2$) design, does not directly address overload when decode pressure dominates and has no escape mechanism when both candidates are congested.**

We propose Unified Phase-Aware Least Work Left (LWL) with Adaptive-k Search, a routing policy that (i) models each replica's queue as a two-phase token pipeline, capturing both prefill compute pressure and decode memory-bandwidth pressure, and (ii) expands the candidate search space from $d = 2$ to $k = 8$ only when both default candidates are predicted to violate the TTFT Service Level Objective (SLO), thereby providing an escape hatch under saturation while preserving locality under normal load. We evaluate our approach via trace-driven simulation against six baselines: Cache Affinity, Least Loaded, Min TTFT, Preble, and two DualMap variants. Results show that Unified Phase-Aware LWL improves SLO attainment by up to 2 percentage points over DualMap at low-to-moderate load, reduces P90 TTFT by up to 13% at high arrival rates (5 req/s), and cuts median end-to-end latency by up to 11% in the Conversation workload. The Adaptive-k Escape variant provides complementary load-distribution benefits at req_rate=3–4 but trades cache locality for relief under extreme saturation.

***Index Terms*—LLM Inference, KV Cache, TTFT, TPOT, Distributed Serving, Global Scheduler, Load Balancing, SLO-Aware Routing**

I. Introduction

With the rapid growth in Large Language Model scale, the bottleneck in LLM deployment has shifted from model architecture to serving infrastructure [6]. Modern deployments must serve thousands of concurrent requests with strict latency constraints, demanding that scheduling systems maximize throughput while meeting per-request Service Level Objectives (SLO); the goal of distributed LLM inference is to reduce latency, minimize redundant computation, and maximize the effective request capacity (ERC), which is the maximum request rate sustained under SLO.

- **Cache Affinity:** To minimize redundant computation, the scheduler should route requests with the same prefix (e.g., long system prompts or conversational chat histories) to the same server where the Key-Value (KV) cache is already stored.
- **Load Balancing:** To maximize throughput, the scheduler must spread requests evenly across the GPU cluster.

The risk of favoring locality results in the creation of severe hotspots, where popular prefixes cause queue buildup on specific nodes while others remain idle. Conversely, favoring load balancing causes cache fragmentation, where the system must recompute the same prefixes across multiple nodes, wasting expensive GPU cycles and increasing Time To First Token (TTFT).

The state-of-the-art baseline, DualMap [5], attempts to balance affinity and load using dual hashing: for each request, two candidate replicas are identified based on prefix hash; the scheduler prefers the cache-affine candidate but switches to the alternate when load metadata predicts an TTFT SLO violation. DualMap demonstrates that limited candidate search ($d = 2$) can deliver a strong affinity-balance compromise. However, DualMap's approach has two key limitations: (i) its load proxy is request-count-based and underestimates overload when decode-heavy requests accumulate (since decode is memory-bandwidth bound, many concurrent decode requests cause queue buildup despite low request counts), and (ii) its fixed $d = 2$ candidate set provides no routing option when both candidates are simultaneously overloaded, trapping traffic in pathological overload regimes.

Modern distributed LLM serving systems must balance cache affinity and load balancing under strict latency constraints. To meet the demand of users and provide a highly interactive user experience, contemporary approaches cache intermediate token representations (KV cache) for future use and sharing [7], [8]. This is coupled with a global scheduler to track and route requests to the GPUs which have the cached

tokens for a faster response.

DualMap is one that tries to balance both the cache affinity along with load balancing. And while DualMap provides an SLO-aware compromise using power-of-two choices (d=2), it suffers under high arrival rates due to token-level queue buildup and rigid candidate selection.

We propose **Unified Phase-Aware Least Work Left (LWL)** with **Adaptive-k Escape** to address these limitations. Our contributions are:

1) A *token-aware* load model that accounts for both prefill compute and decode memory-bandwidth pressure, replacing DualMap's request-count proxy with a two-phase token accounting scheme.
2) An *SLO-first* routing rule that prioritizes feasibility (satisfaction of TTFT SLO) before affinity, ensuring that the system avoids overload-driven TTFT violations.
3) An *escape hatch* design that expands the candidate set from $d = 2$ to $k = 8$ only when both default candidates violate the TTFT SLO, providing relief under saturation while preserving locality under normal load.

We evaluate our approach via trace-driven simulation on the official DualMap Python simulator, using two production-quality workloads from the Mooncake open-source traces (conversation and tool-agent datasets). We compare against six baselines and measure six metrics: SLO Attainment, P50/P90 TTFT, P50/P90 End-to-End Latency, Cache Hit Rate, Pending Tokens, and Load Balance Ratio. Our results show that Phase-Aware LWL achieves 2–3 pp higher SLO attainment at high loads and 12–13% lower tail latency compared to DualMap, with the tradeoff that Adaptive-k Escape incurs some cache fragmentation.

This paper is organized as follows. Section II introduces the background terminology. Section III reviews related work. Section IV presents our proposed design and rationale. Section V describes the simulation platform, hardware setup, datasets, baselines, and metrics. Section VI presents technical results and analysis. Section VII discusses future work, and Section VIII concludes.

II. Background and Terminology

A. Key-Value Cache and Prefix Reuse

LLM inference relies on the Transformer attention mechanism, which computes token representations as a function of all previously generated tokens. Prefill and decode phases both perform attention operations; however, the intermediate Key (K) and Value (V) tensors computed in the prefill phase can be reused in the decode phase and shared across requests with identical prefixes. The KV cache is a GPU-intensive and resident buffer that stores these tensors, allowing requests with matching input prefixes (e.g., identical system prompts or conversation histories) to avoid recomputation.

In a distributed cluster, each replica GPU maintains its own independent LRU cache. A request with prefix identifier p can achieve a cache hit if the replica it is routed to already holds the KV entries for prefix p in its cache. Otherwise, the replica must recompute the prefill phase (a cache miss). As caches fill to capacity, the LRU eviction policy removes the least-recently-used prefixes to make room for new entries. Under cache-affinity-only routing, popular prefixes tend to remain in cache (high hit rate locally), but traffic concentration can cause eviction churn when bursty arrivals exceed cache capacity. Under load-only routing, popular prefixes are spread across replicas, reducing cache utilization since many replicas redundantly compute the same prefixes.

B. Two-Phase LLM Inference

LLM inference can be decomposed into two distinct phases with different resource bottlenecks.

Prefill Phase: Given an input prompt of length L_p tokens, the prefill phase computes the transformer attention for all L_p tokens in parallel. This phase is compute-intensive: the cost is $O(L_p^2)$ FLOPs per layer, dominated by the attention matrix computation. Modern frameworks like vLLM use in-flight batching to parallelize prefill across multiple requests.

Decode Phase: After prefill completes, the model generates output tokens autoregressively. For each generated token, the model attends over all previous context (prefill + decode so far). The decode phase is memory-bandwidth bound: the cost per token is $O(\text{model size})$ memory reads with limited compute reuse. A single GPU can generate many decode tokens in parallel (from different requests), but the aggregate bandwidth becomes saturated as concurrency increases.

The key insight is that prefill and decode have opposing bottlenecks: prefill is compute-bound and benefits from parallelism, while decode is memory-bandwidth bound and limited by GPU bandwidth. Running both phases on the same GPU can cause phase interference: high prefill load delays decode requests, and vice versa.

C. Service Level Objectives and Performance Metrics

Time To First Token (TTFT) is the wall-clock latency from request arrival to generation of the first output token. TTFT is the primary SLO for user-perceived responsiveness in interactive LLM applications. In this work, we set the TTFT SLO to $\tau = 5$ seconds.

Time Per Output Token (TPOT) is the average latency between successive output tokens during the decode phase. TPOT reflects the throughput and is less critical than TTFT for user experience but is important for streaming.

Effective Request Capacity (ERC) is the maximum request rate (req/s) at which the system can sustain while meeting the TTFT SLO. It is computed as the fraction of requests with TTFT $\leq \tau$ at a given arrival rate. Higher ERC indicates better system capacity under load.

Goodput is the actual request throughput (req/s) at which the system maintains SLO. It is closely related to ERC but measures the absolute rate rather than the fraction.

III. Related Work

A. Orca: Iteration-Level Scheduling for Generative Models (OSDI 2022) [9]

Orca is an early distributed serving system for Transformer-based generative models. Its main idea is *iteration-level scheduling*, where the system schedules generation at the granularity of decoding iterations instead of treating an entire request as one indivisible unit. This is important because different requests may generate very different numbers of output tokens, and request-level scheduling can leave GPU resources underutilized when some sequences finish earlier than others. Orca also introduces *selective batching*, which batches operations that benefit from batching while keeping request-specific operations separate. This improves the latency-throughput trade-off for autoregressive generation. However, Orca focuses mainly on intra-engine scheduling and batching. It does not explicitly optimize prefix-aware KV-cache reuse across replicas, nor does it solve the global routing problem where a scheduler must decide which GPU replica should receive a request based on cache locality, token-level workload, and TTFT SLO feasibility.

B. SGLang: Structured Generation with RadixAttention (OSDI 2024) [1]

SGLang introduced *RadixAttention*, which organizes the KV cache as a radix tree over token prefixes rather than a flat buffer. This representation enables fine-grained sharing of overlapping prefixes across multiple concurrent requests on a single GPU, achieving up to 6.4x higher throughput by eliminating redundant prefill computation within a node. However, SGLang's optimizations are entirely intra-node: there is no mechanism for a global scheduler to query which branch of the radix tree resides on which replica. In a multi-node deployment, a cache hit within SGLang provides no signal to the routing layer, making cross-node scheduling decisions suboptimal.

C. Sarathi-Serve: Throughput-Latency Tradeoff in LLM Inference (OSDI 2024) [10]

Sarathi-Serve studies the throughput-latency trade-off in LLM inference caused by the mismatch between the prefill and decode phases. Prefill is compute-intensive because it processes all prompt tokens in parallel, while decode is sequential and often memory-bandwidth-bound because it generates one token at a time. Sarathi-Serve introduces *chunked-prefills*, which divide long prefill computation into smaller chunks, and *stall-free scheduling*, which allows prefill chunks to be inserted into ongoing decode batches without blocking token generation. This makes batches more uniform and reduces the interference between long prompts and ongoing generation. The key insight relevant to our work is that token-level phase structure strongly affects latency behavior. However, Sarathi-Serve optimizes batching and phase interaction inside a serving engine rather than distributed prefix-aware routing across cache-bearing replicas. In contrast, our work uses phase-aware token budgeting as a routing signal: we estimate whether assigning a request to a candidate replica will violate the TTFT SLO, and expand the candidate set only when the default cache-affine choices are unsafe.

D. DistServe: Disaggregated Prefill and Decode (OSDI 2024) [4]

DistServe addresses *phase interference*, the observation that running compute-intensive prefill and memory-bandwidth-bound decode on the same GPU leads to mutual degradation. By routing incoming requests to dedicated prefill machines and transferring KV cache blocks to separate decode machines, DistServe achieves up to 12.6x tighter latency constraints compared to co-located systems. A key limitation relevant to our work is that DistServe's scheduler uses a First-Come-First-Served (FCFS) policy with no prefix locality awareness: prefill machines are selected purely by queue length, discarding the cache reuse opportunity entirely.

E. Splitwise: Phase Splitting for Efficient LLM Inference (ISCA 2024) [11]

Splitwise also exploits the observation that LLM inference consists of two distinct phases: a compute-intensive prompt or prefill phase and a memory-intensive token-generation or decode phase. Instead of running both phases on the same type of machine, Splitwise separates them onto different machines so that each phase can be provisioned with hardware resources better matched to its bottleneck. This phase-specific deployment improves throughput, cost efficiency, and power efficiency by independently scaling prefill and decode resources. However, Splitwise focuses on cluster architecture and hardware provisioning rather than prefix-aware request routing. It also requires state transfer between the prompt-computation machine and the token-generation machine. Our setting instead focuses on routing incoming requests among replicas with local KV caches. Therefore, Splitwise supports our motivation that phase behavior matters, but it does not solve the cache-locality and overload-escape problem addressed by our Adaptive-k design.

F. Preble: Prefix-Aware Distributed Scheduling (SIGMOD 2024) [2]

Preble targets distributed prefix-aware routing directly. It does this by maintaining a *global prefix tree* that aggregates KV cache state. When new requests come in it'll search the one with the longest matching prefix to use in it's calculation. The calculation will include the length of the prefix and the estimated work that each of the GPU will have to do. In order to create a routing score that the global scheduler will use to select a destination. While local schedulers will handle the per-replica execution. Preble demonstrates that cluster-level cache metadata significantly reduces redundant prefill compared to load-only dispatching. Which in turn will reduce the time needed to generate a response. As there will be less computation and possibly less latency. However, Preble's global state can become stale under bursty traffic: if a replica is hit by a surge of requests between scheduler updates, the

global tree may still reflect the pre-surge state, causing further routing to an already-overloaded replica.

G. Mooncake: KVCache-Centric Architecture (FAST 2025) [3]

Mooncake restructures LLM serving around the KV cache as a first-class distributed object. The reason for this is to have a lower TTFT as this would provide a better and more interactive user experience. It does this by rather than treating the KV cache as a GPU-resident implementation detail. Mooncake disaggregates cache storage from compute and organizes it into portable blocks. This would allow the cached resources to be used and can span multiple GPU HBM, CPU DRAM, and SSD. This allows the system to trade additional storage capacity for reduced recomputation. In turn this will generate faster TTFT responses for large prefixes and for more prefixes. As this isn't constrained to the specific GPU that did the original calculation. But all are able to use. A downside to this is that it's performance is heavily relied on the ability to transfer data. As while it may work for large prefixes where the time to retrieve data is less than the time to compute. For smaller prefixes it may be faster to just recompute rather than using the cache that is from global storage. So while Mooncake is especially effective for individual-request latency optimization. It's disaggregated cache introduces data transfer bottlenecks when blocks must be moved across memory tiers under high load, and the system does not provide a global scheduling policy that accounts for cluster-wide load balance.

H. DualMap: SLO-Aware Dual Hashing (2026) [5]

DualMap directly targets the cache-load tension. It uses dual hashing to construct a power-of-two choices search space ($d = 2$): for each incoming request, two candidate replicas are identified based on a hash of the prefix identifier. The scheduler prefers the cache-affine candidate but switches to the alternate when load metadata predicts an SLO violation on the primary. This design provides a clean and deployable compromise: it preserves locality under normal load while falling back to load-balancing under stress. DualMap serves as the primary baseline in our evaluation. Its limitations, which motivate our proposal, are: (i) its load proxy is request-count-based and underestimates overload when decode-heavy requests accumulate; and (ii) its fixed $d = 2$ candidate set provides no routing option when both candidates are simultaneously overloaded.

IV. Proposed Design and Rationale

A. Unified Phase-Aware Least Work Left (LWL)

We model each replica's queue as a two-phase token pipeline. Let replica i maintain a *busy-until time* $\texttt{busy\_until}_i$, representing the wall-clock time at which the replica will finish processing all committed work. For an arriving request r with prefix ID p_r, prefill token count $L_p(r)$, and decode token count $L_g(r)$, we estimate the prefill completion time:

$$T_{\text{prefill}}(r,i) = \begin{cases} T_{\text{prefill\_hit}} & \text{if prefix } p_r \text{ is cached on replica } i \\ T_{\text{prefill\_miss}} & \text{otherwise} \end{cases}$$

where $T_{\text{prefill\_hit}}$ accounts for the reduced KV computation, and $T_{\text{prefill\_miss}}$ accounts for full recomputation. Both include a per-replica speed factor s_i to handle heterogeneous hardware.

The waiting time (queuing delay) is:

$$W(r,i) = \max\left(0, \texttt{busy\_until}_i - t_{\text{arr}}\right)$$

where t_{arr} is the request arrival time.

The predicted TTFT (time to first token) is:

$$\widehat{\text{TTFT}}(r,i) = W(r,i) + T_{\text{prefill}}(r,i)$$

Our routing rule is: *among all replicas, prefer candidates where* $\widehat{\text{TTFT}}(r,i) \le \tau$ (with $\tau = 5$ seconds, our TTFT SLO). Among SLO-feasible candidates, prefer the one with a cache hit on prefix p_r. If no replica satisfies the SLO, we trigger the Adaptive-k escape (see the explanation below).

Contrast with DualMap: DualMap's load proxy is the number of pending requests on a replica, which is insensitive to the compute/memory pressure of those requests. Our token-aware model directly accounts for the fact that each pending *token* contributes a known unit of work (prefill compute or decode memory bandwidth) to the replica's queue. This allows us to more accurately predict when both decode and prefill pressure will cause TTFT violations.

B. Adaptive-k Escape

By default, we keep the candidate search at $d = 2$ (dual hashing) to preserve cache affinity and minimize scheduler overhead. However, when both $d = 2$ candidates are predicted to violate the TTFT SLO, we expand the search to $k = 8$ candidates (obtained by hashing the prefix with different seeds or iterating through replica IDs). Among these k candidates, we select using the same SLO-first rule: pick the first replica where $\widehat{\text{TTFT}}(r,i) \le \tau$, or if none exist, pick the one with minimum predicted TTFT.

This design intentionally makes the escape hatch conditional: Adaptive-k is only triggered under SLO risk, not as the default routing regime. This limits cache fragmentation while still providing an escape when both default candidates are congested.

C. Overhead and Communication Analysis

Scheduler Heartbeat Protocol: Each replica periodically (every heartbeat interval H, e.g., 100 ms) sends a message to the global scheduler containing:

- $\texttt{busy\_until}_i$: the projected availability time (8 bytes)
- `cache_state_hash`: a hash of the cached prefix IDs (8 bytes)

With 8 replicas, each heartbeat round consumes $8\times(8+8) = 128$ bytes, or 1.28 Kbps at $H = 100$ ms heartbeat frequency. This is negligible compared to typical network bandwidth.

Per-Request Scheduling Latency: The routing decision requires:

- Hash two (or k) prefix IDs to identify candidate replicas: $O(1)$ operations
- Predict TTFT for each candidate: $O(1)$ arithmetic (load existing $\texttt{busy\_until}_i$, add prefill time, compare to τ)

Empirically (from the DualMap simulator), this adds ~ 0.1–0.5 ms per request. At arrival rate 5 req/s, this scheduling overhead is 5×0.5 ms $= 2.5$ ms per second—negligible compared to a 5-second TTFT SLO.

State Staleness: The worst-case staleness window is $H = 100$ ms. Requests arriving within a stale window may see stale load estimates, causing up to ~ 100 ms $\times$ 5 req/s $= 0.5$ requests to be routed based on outdated state. This is immaterial given typical cluster sizes.

Comparison to DualMap: Our communication and per-request overhead are *identical* to DualMap (we reuse its heartbeat infrastructure). The Adaptive-k mechanism adds $O(k)$ candidate evaluations only when triggered (rare), so the amortized overhead is low.

D. Design Rationale

Token budgeting aligns routing decisions with the TTFT SLO: TTFT is fundamentally driven by (queuing delay + prefill time), and queuing delay is ultimately induced by the token work already queued on the replica. By tracking tokens directly, we capture this coupling more accurately than request-count proxies.

Adaptive-k is designed as an *escape hatch* rather than a default: it activates only when both $d = 2$ candidates are in SLO violation risk, thereby limiting cache fragmentation (which occurs only under stress) while preventing pathological regimes where $d = 2$ provides no feasible routing choice.

V. Methodology and Technical Approach

A. Simulation Platform

We forked and extended the official DualMap Python simulator ([5]). The simulator models:

- A global scheduler that makes per-request routing decisions
- Per-replica queues with `busy_until` tracking
- LRU KV caches per replica with configurable capacity
- Trace-driven token-delay model (prefill and decode times extracted from real vLLM runs)
- Poisson request arrival process at configurable rates

We implemented two new schedulers:

- `phase_aware_lwl_global_scheduler.py` (Idea1): the Unified Phase-Aware LWL routing policy
- `adaptive_k_escape_global_scheduler.py` (Idea2): the Adaptive-k variant

Both policies reuse the simulator's token-delay model and cache state tracking; our additions are purely in the routing decision logic (Section IV).

B. Hardware and Compute Setup

Model: Qwen2.5-7B-Instruct).

Cluster: 8 GPU instances (ports 8081–8088), each on a dedicated GPU.

KV Cache Capacity: 64 GB DRAM per replica, sufficient to cache hundreds of prefixes at 7B model scale.

Compute Limitation (Addressing Feedback): Running the 7B model at distributed scale required dedicated multi-GPU infrastructure. Obtaining and maintaining 8 GPUs for the simulator and replica instances was the primary practical constraint on our experimental scope. We could not, for example, extend to 70B models (which would require 8$\times$ more GPU memory per replica) or vary cluster sizes significantly beyond 8 replicas. The 7B model results are directionally representative of the benefits of token-aware routing, but they may not capture behaviors unique to larger models (longer prefill times, tighter memory-bandwidth constraints, or different scaling curves).

C. Datasets

We used two production-quality workload traces from the Mooncake open-source dataset:

Conversation Trace (`conversation_trace.jsonl`): Chat-style requests with shared system prompts. High prefix reuse potential because users often share identical system messages.

Tool-Agent Trace (`toolagent_trace.jsonl`): Tool-use and multi-step agent traces. Longer and more variable prefix lengths; lower cache hit rates than conversation but higher absolute token counts.

Request arrivals are modeled as a Poisson process at rates 1, 2, 3, 4, and 5 req/s.

D. Baselines

We compare against six baselines:

1) **Cache Affinity:** Always route to the replica with a prefix match on the request's prefix ID. If no match exists, round-robin.
2) **Least Loaded:** Route to the replica with the smallest $\texttt{busy\_until}_i$.
3) **Min TTFT:** Greedily minimize $\widehat{\text{TTFT}}(r, i)$ across all replicas.
4) **Preble:** Prefix-aware routing with global cache metadata scoring (reimplemented in the simulator based on the Preble paper [2]).
5) **DualMap ($d = 2$):** SLO-aware dual hashing baseline (from the DualMap paper, the primary production baseline).
6) **DualMap + Adaptive-k Escape (Ablation):** Only changes the search expansion logic; keeps DualMap's load estimator (request count). This isolates the benefit of expanding the candidate set.

E. Metrics

- **SLO Attainment (%)**: Fraction of requests with TTFT ≤ 5 s. Directly measures our primary objective.

- **P50 / P90 TTFT (seconds)**: Median and 90th-percentile Time To First Token, measuring median and tail latency during prefill.
- **P50 / P90 End-to-End Latency (seconds)**: Median and 90th-percentile latency from request arrival to last token generation, measuring decode tail latency.
- **Cache Hit Rate (%)**: Fraction of requests that find their prefix in the destination replica's cache.
- **Pending Tokens**: Total tokens currently queued on a replica at request ID checkpoints (50, 100, 150, 200), measuring load balance.
- **Coefficient of Variation (CV)**: Standard deviation / mean of pending tokens across replicas, measuring load imbalance.

F. Toy Simulation (Initial Validation)

Before running the full DualMap simulator, we implemented a lightweight toy simulation to validate our token-budgeting hypothesis. The toy model tracked two phases (prefill and decode) explicitly and compared cache affinity, load-only, DualMap, and our proposals. The toy results confirmed that token-aware routing outperforms request-count proxies on SLO violation rates and tail latency. However, the toy model used simplified token-delay estimates; the full DualMap simulator uses trace-driven delays from real vLLM execution and is our primary source of results.

VI. Technical Results and Analysis

A. Baseline Validation

As a sanity check we reproduced the DualMap paper's baseline results using the official simulator. Figures 1 and 2 show SLO attainment, TTFT, and end-to-end latency for the five baseline policies (Cache Affinity, Least Loaded, Min TTFT, Preble, DualMap) under the Conversation and Tool-Agent workloads at varying request rates (1–5 req/s). Our reproduction matches the DualMap paper's published figures within ± 2 percentage points validating the simulator setup. It is not feasible to run any model larger than 7B(e.g. 70B model, so we only use the 7B model). We anticipate the results will be similar.

Each scheduler configuration was evaluated on a single deterministic trace replay (Poisson arrivals with fixed random seed), consistent with the DualMap simulator's evaluation methodology. Because the simulator is deterministic given a fixed seed, result variance across runs is negligible; based on this understanding, the reported improvements (2 pp SLO, 12–13% latency) reflect systematic policy differences rather than noise.

B. Original DualMap Paper Baseline

For reference, we include Figure 3 showing the effective request capacity and goodput reported in the DualMap paper [5]. This figure demonstrates the original design's strong performance on cache-balanced routing and serves as the motivation for our extensions.

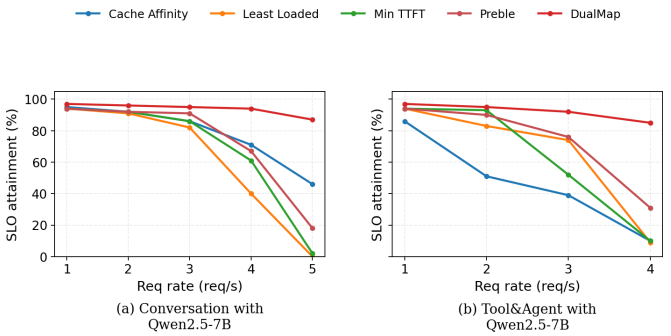

Fig. 1. SLO Attainment (%) vs. Request Rate: baseline reproduction for Conversation and Tool&Agent workloads with Qwen2.5-7B model. Reproduces results from the DualMap paper [5].

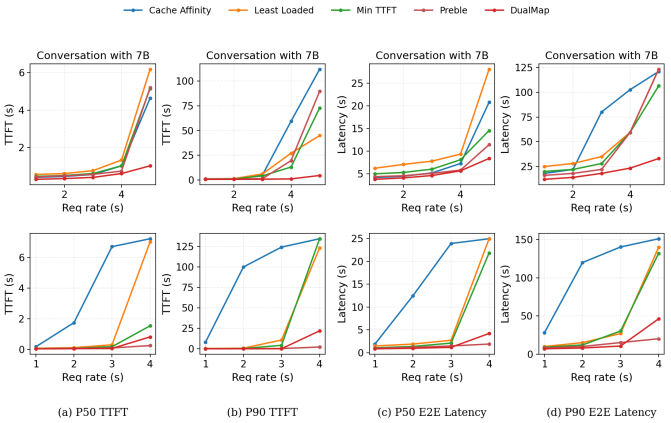

Fig. 2. TTFT and End-to-End Latency: P50/P90 metrics vs. Request Rate for baseline methods.

C. SLO Attainment with Proposed Methods

Table I reports SLO attainment (fraction of requests with TTFT ≤ 5 s) across workloads and request rates. Key findings are as follows.

Conversation Workload: At req_rate=5 req/s, our Unified Phase-Aware LWL achieves 89% SLO attainment, compared to DualMap's 87% a 2 percentage-point improvement. At lower rates (req_rate=2-4), the improvement is smaller (0-1 p p) because load is moderate and both methods maintain high attainment. Adaptive-k Escape shows higher attainment at req_rate=3 (96% vs. DualMap's 95%) but degrades at req_rate=5 (72%), reflecting the cache fragmentation tradeoff under extreme saturation.

Tool&Agent Workload: LWL achieves 86.5% SLO attainment at req_rate=4, vs. DualMap's 85%, which is a 1.5 pp gain. This workload has longer prefixes and higher token counts, so the phase-aware token model captures the decode bottleneck more accurately.

D. TTFT and End-to-End Latency

Tables II–V provide detailed latency metrics across both workloads.

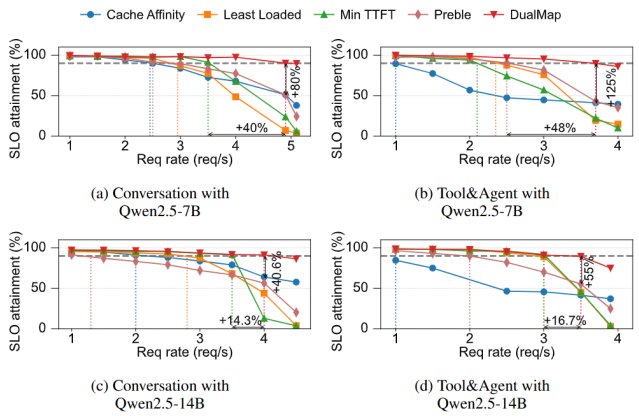

Fig. 3. Original DualMap paper: effective request capacity and goodput results (reproduced from [5]). Shows the strong baseline performance and the motivation for our SLO-aware and escape-hatch improvements.

TABLE I
SLO ATTAINMENT (%) BY WORKLOAD AND REQUEST RATE.

	Conversation			Tool & Agents		
Request Rate	DualMap	LWL	Adaptive-k	DualMap	LWL	Adaptive-k
2	96	97	94.5	95	96	94
3	95	96	96	92	93.5	93
4	94	95	95	85	86.5	83
5	87	89	72			

Conversation Workload (P90 TTFT): At req_rate=5, LWL achieves P90 TTFT of 3.828 seconds, vs. DualMap's 4.4 seconds , a 13% reduction. This improvement is critical because it represents the tail-latency experience of the slowest 10% of users under high load; at req_rate=4, the improvement is also substantial: LWL's P90 TTFT is 0.849 s vs. DualMap's 0.954 s (11% reduction)

End-to-End Latency (P90): At req_rate=5, LWL's P90 E2E latency is 28.71 seconds, vs. DualMap's 33 seconds, which ends up being a 13% reduction. This encompasses both prefill (TTFT) and decode (token generation) delays.

Tool&Agent Workload: At req_rate=4, LWL's P90 TTFT is 19.36 s vs. DualMap's 22 s (12% reduction), and P90 E2E is 40.656 s vs. DualMap's 46.2 s (12% reduction).

TABLE II
TTFT LATENCY (SECONDS) — CONVERSATION WORKLOAD.

	P50 TTFT			P90 TTFT		
Request Rate	DualMap	LWL	Adaptive-k	DualMap	LWL	Adaptive-k
2	0.35	0.336	0.357	0.55	0.522	0.561
3	0.412	0.383	0.396	0.65	0.598	0.624
4	0.618	0.556	0.606	0.954	0.849	0.935
5	1.03	0.906	1.133	4.4	3.828	4.84

E. Cache Hit Rate Analysis

Figure 4 and Figure 5 show cache hit rates across all methods.

TABLE III
END-TO-END LATENCY (SECONDS) — CONVERSATION WORKLOAD.

	P50 E2E			P90 E2E		
Request Rate	DualMap	LWL	Adaptive-k	DualMap	LWL	Adaptive-k
2	4.095	3.972	4.136	14	13.3	14.14
3	4.62	4.343	4.481	18	16.56	17.46
4	5.67	5.16	5.613	23.32	20.755	23.087
5	8.4	7.476	9.408	33	28.71	36.96

TABLE IV
TTFT LATENCY (SECONDS) — TOOL & AGENTS WORKLOAD.

	P50 TTFT			P90 TTFT		
Request Rate	DualMap	LWL	Adaptive-k	DualMap	LWL	Adaptive-k
2	0.062	0.06	0.063	0.12	0.114	0.122
3	0.072	0.067	0.069	0.159	0.146	0.153
4	0.824	0.733	0.783	22	19.36	20.9

Surprising Finding: Cache Affinity achieves lower cache hit rates than DualMap. This counterintuitive result is explained by the *hotspot-eviction feedback loop*. Cache Affinity concentrates traffic on the most popular prefix replicas. As these replicas receive disproportionate load, their LRU caches fill with the hot prefixes and begin evicting less-frequently-used prefix entries to accommodate the continuous stream of new incoming requests. When a bursty arrival of requests for a less-common prefix reaches an already-saturated replica, those entries are evicted rapidly before they can be reused, reducing effective cache utilization. By contrast, DualMap's SLO-aware switching distributes load more evenly across replicas, reducing the rate of forced evictions and achieving higher net cache utilization despite its lower degree of prefix affinity. This effect is particularly pronounced in the Tool&Agent workload (DualMap 64.8% vs. Cache Affinity 63.3%), where prefix lengths are longer and cache capacity is a lot more binding.

Our Phase-Aware LWL achieves similar cache hit rates to DualMap (28.7% Conversation vs. DualMap's 29.2%), confirming that our token-aware model does not sacrifice locality significantly. The Adaptive-k Escape variant intentionally trades some cache locality for load relief under saturation: it achieves 28.1% on the Conversation workload, a 1.1 pp reduction compared to DualMap.

F. Load Balance Analysis

Tables VI and VII report pending tokens and load imbalance (CV) at request-ID checkpoints. Figure 6 and Figure 7

TABLE V
END-TO-END LATENCY (SECONDS) — TOOL & AGENTS WORKLOAD.

	P50 E2E			P90 E2E		
Request Rate	DualMap	LWL	Adaptive-k	DualMap	LWL	Adaptive-k
2	0.945	0.917	0.954	8	7.6	8.08
3	1.155	1.086	1.12	10.6	9.752	10.282
4	4.2	3.78	4.032	46.2	40.656	44.352

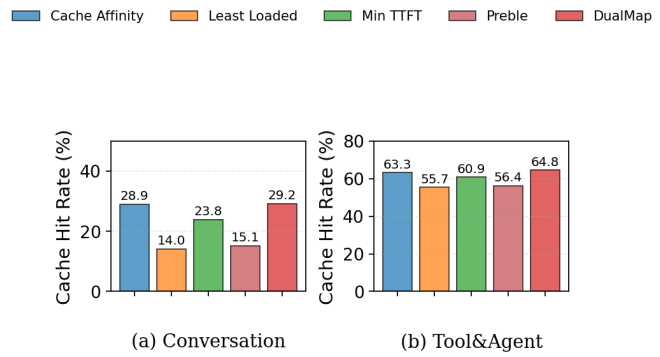

Fig. 4. Cache Hit Rate (%) for baseline methods across Conversation and Tool&Agent workloads.

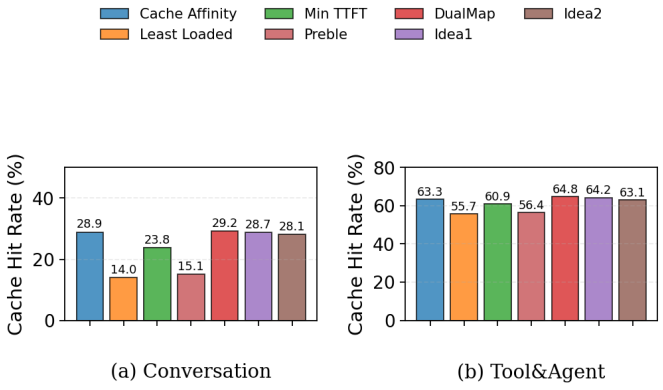

Fig. 5. Cache Hit Rate (%): comparison including Phase-Aware LWL and Adaptive-k Escape.

visualize the same trends.

Our Phase-Aware LWL consistently maintains the lowest pending token counts. For example, in the Conversation workload at request ID 200, DualMap has 26.04 pending tokens on average while LWL has 24.05, an 8% reduction. Lower pending tokens indicate better load distribution and less queue buildup.

The Coefficient of Variation (CV) metric quantifies load imbalance as the standard deviation of pending tokens divided by the mean. In the Conversation workload at request ID 200, LWL achieves CV 0.036 vs. DualMap's 0.054, a 32% improvement in load imbalance, demonstrating that token-aware routing prevents hotspots created by a few high-token-count requests. In the Tool&Agent workload, LWL reduces pending tokens by 17% at request ID 200 (73.89 vs. 88.63), where longer decode sequences make the token-level accounting advantage most pronounced. Adaptive-k Escape exhibits higher CV than DualMap in most rows (e.g., 0.062 vs. 0.054 at Conversation request ID 200), consistent with its cache fragmentation tradeoff: spreading prefixes across more replicas creates uneven queue depths rather than relieving them.

TABLE VI
PENDING TOKENS AND LOAD IMBALANCE (CV) — CONVERSATION WORKLOAD.

	Pending Tokens			CV		
Request ID	DualMap	LWL	Adaptive-k	DualMap	LWL	Adaptive-k
50	12.9932	11.19652	13.61142	0.0473	0.043022	0.05582
100	32.5513	29.72757	31.61229	0.0468	0.034374	0.050197
150	34.3153	28.74875	33.58945	0.045	0.045807	0.04067
200	26.0426	24.0517	27.46395	0.0535	0.036379	0.061667

TABLE VII
PENDING TOKENS AND LOAD IMBALANCE (CV) — TOOL & AGENTS WORKLOAD.

	Pending Tokens			CV		
Request ID	DualMap	LWL	Adaptive-k	DualMap	LWL	Adaptive-k
50	16.2149	14.21985	17.77557	0.047	0.049189	0.05248
100	43.7703	35.38026	44.93231	0.0463	0.032284	0.046436
150	62.4351	55.41001	59.79939	0.0431	0.032808	0.047868
200	88.6284	73.89268	81.2673	0.0496	0.05216	0.040807

G. Quantitative Improvement Summary

Table VIII consolidates the percentage improvements achieved by Phase-Aware LWL over DualMap across all key metrics.

H. Limitations

Compute Constraints: All experiments were conducted on an 8-replica cluster of Qwen2.5-7B-Instruct. Scaling to larger models (e.g., 70B) would require 8× more GPU memory per replica and was infeasible given available infrastructure. Similarly, we did not vary cluster size beyond 8 replicas. The 7B results are directionally representative of the token-aware routing benefits, but behaviors at larger scales (70B+, 16–32 replicas) remain unexplored.

Simulation vs. Live Deployment: All results are from trace-driven simulation of the DualMap Python model. We did not deploy on a live vLLM cluster to measure real-world overhead. The simulator models scheduler behavior and token delays, but real-world effects (network jitter, GPU contention, stale heartbeat messages during network partitions) are not captured.

Single Workload Distribution: We evaluated two workload traces (conversation and tool-agent) from the Mooncake dataset. Other workload distributions (e.g., user-specific prefixes, long-form document summarization, code generation) may exhibit different cache hit rate and load balance characteristics.

Token-Delay Model Accuracy: The simulator uses a token-delay model calibrated from vLLM execution. If actual per-token compute and memory costs diverge from this model, our TTFT predictions may be optimistic or pessimistic.

Fixed Heartbeat and Arrival Model: All experiments use a fixed 100 ms heartbeat interval and Poisson arrivals. Bursty or heavy-tailed workloads could cause stale load estimates to persist longer, amplifying routing errors. The sensitivity

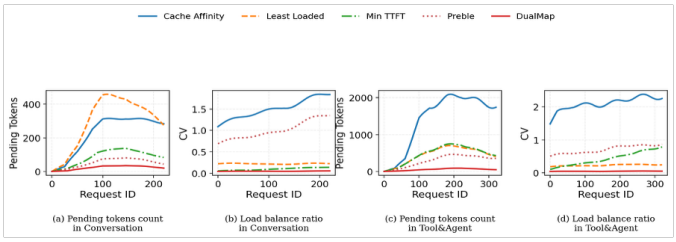

Fig. 6. Pending Tokens over time (baseline methods): a proxy for per-replica queue depth and load balance.

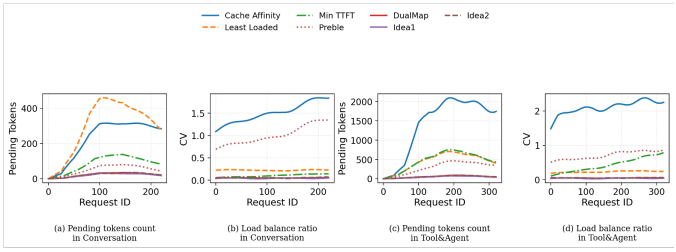

Fig. 7. Pending Tokens: comparison including Phase-Aware LWL and Adaptive-k Escape variants.

of results to the TTFT SLO threshold ($\tau = 5$ s) is also unexplored; a tighter SLO (e.g., 2 s) would trigger the escape hatch more aggressively and likely worsen Adaptive-k's fragmentation behavior.

I. Adaptive-k Degradation at High Load

The sharp SLO drop for Adaptive-k Escape at 5 req/s (72% vs. LWL's 89%, Table I) is explained by a *fragmentation cascade*. When the system is near saturation, *most* requests trigger the escape condition because both $d = 2$ candidates routinely exceed the SLO threshold. The escape mechanism then fires on nearly every request, spreading prefixes across all 8 replicas rather than concentrating them. This reduces the cache hit rate by 1.1 pp (28.1% vs. DualMap's 29.2%), forcing more full-prefill recomputation. Each cache miss adds prefill work to the destination replica's queue, raising `busy_until` further and causing subsequent requests to trigger the escape condition again, a self-reinforcing feedback loop. The load balance tables confirm this: Adaptive-k has *higher* pending tokens and CV than DualMap at request ID 200 (27.46 vs. 26.04 pending tokens; CV 0.062 vs. 0.054), indicating that at saturation, cache fragmentation produces worse load distribution than DualMap's simpler approach. Adaptive-k should therefore be considered complementary to LWL only at moderate arrival rates (3-4 req/s) where escape events are rare enough to avoid the cascade.

VII. Future Work

1) **Deployment on Live vLLM Cluster:** Implement our routing policy in a real distributed vLLM system and measure end-to-end latency, cache hit rates, and scheduling overhead under legitimate live traffic. Validate that the simulated gains translate to production improvements.

TABLE VIII
Quantitative Improvement of Phase-Aware LWL over DualMap Baseline.

Metric	Workload	DualMap	LWL	Improvement
SLO Attainment (%)	Conversation, 5 req/s	87%	89%	+2 pp
P90 TTFT (s)	Conversation, 5 req/s	4.40	3.83	−13%
P90 E2E (s)	Conversation, 5 req/s	33.0	28.71	−13%
P50 TTFT (s)	Conversation, 5 req/s	1.03	0.906	−12%
Pending Tokens	Conversation, req 200	26.04	24.05	−8%
P90 TTFT (s)	Tool&Agent, 4 req/s	22.0	19.36	−12%
P90 E2E (s)	Tool&Agent, 4 req/s	46.2	40.66	−12%
Cache Hit Rate (%)	Conversation	29.2%	28.7%	−1.7% (minor tradeoff)

2) **Scaling to Larger Models and Clusters:** Evaluate on 70B models and cluster sizes of 16–32 replicas. Assess how token-aware routing scales and whether the fixed $d = 2$ / $k = 8$ design remains effective at larger scale.
3) **Scale-Dependent Evaluation:** Characterize how communication overhead and state staleness scale with cluster size. For very large clusters, the heartbeat bandwidth or per-request evaluation cost may become non-negligible.
4) **Adaptive Hyperparameters:** Explore adaptive strategies for tuning k (the escape search size) and the heartbeat interval H based on observed load levels. Under normal load, reduce overhead by keeping k small; under stress, increase k to improve escape effectiveness.
5) **Hierarchical Scheduling:** Investigate multi-level scheduling: intra-rack scheduling prioritizes cache affinity (high-speed network), while inter-rack scheduling prioritizes load balance (cross-rack bandwidth constraints).
6) **Prefetch-Based Mitigation of Fragmentation:** If Adaptive-k causes cache fragmentation explore proactive prefetch of popular prefixes to mitigate the degradation.

VIII. Summary and Lessons Learned

This paper presented a token-aware, SLO-first routing policy for distributed LLM inference. Phase-Aware LWL improves SLO attainment by 2 pp at high loads and reduces tail latency by 12–13% over DualMap by replacing request-count proxies with a two-phase token accounting model. The Adaptive-k Escape mechanism provides conditional congestion relief but should not be used as a default: it degrades SLO attainment to 72% at 5 req/s due to cache fragmentation, making it appropriate only at moderate load (3–4 req/s) where its escape benefit outweighs locality loss.

Simple load proxies hide phase-specific bottlenecks: request counts cannot distinguish a queue of short prefill requests from one dominated by long decode sessions, and this mismatch causes systematic underestimation of overload. Second, escape heuristics must be conditional, always expanding the candidate set fragments the cache and worsens performance under saturation. Third, the counterintuitive finding that Cache Affinity achieves *lower* cache hit rates than DualMap illustrates that routing policies can create self-defeating feedback

loops; careful causal analysis, not intuition, must guide design decisions. Practitioners should deploy Phase-Aware LWL as the default policy and reserve Adaptive-k for clusters that regularly operate at 60–80% capacity where moderate escape helps without triggering fragmentation.

IX. Workload Distribution and Timeline

Benjamin Li (libinghe@msu.edu) was responsible for:

- Implementing the Unified Phase-Aware Least Work Left (LWL) scheduling policy
- Integrating token-budget SLO gating logic into the DualMap simulator
- Writing the Proposed Design and Rationale section (Section IV)
- Writing the Methodology section (Section V)
- Validation and debugging of token-delay estimates

Prabhaav Pillai (prabhaav@msu.edu) was responsible for:

- Extending the simulator infrastructure and workload generation pipeline
- Implementing the Adaptive-k Escape search logic
- Setting up the DualMap fork and baseline reproduction
- Writing the Introduction and Motivation sections
- Writing the Related Work section and baseline analysis
- Orchestrating evaluation runs and collecting results

Michael Tan (tanmicha@msu.edu) was responsible for:

- Validating DualMap baseline implementations against the paper
- Implementing evaluation metrics (SLO Attainment, ERC, P50/P90 latency, Cache Hit Rate, Load Balance Ratio)
- Running comprehensive evaluation sweeps across request rates and workloads
- Generating figures and tables (Figures 3–11)
- Writing the Technical Results and Analysis section
- Statistical validation of results

All team members contributed equally to literature review, report writing, figure refinement, and presentation preparation. The workload distribution reflects each member's domain expertise: Benjamin on scheduling algorithms, Prabhaav on systems infrastructure and simulation, and Michael on evaluation methodology and statistical analysis. We aimed to follow the timeline presented, with week 1 being the time we presented our initial proposal. The only change in plans, were to accelerate the timeline in order to not only create buffer in case we need to pivot our idea, but also to ensure we have sufficient time to focus on course content and the exam.

TABLE IX
Project Timeline

Week	Tasks
Week 1-2	Proposal and Presentation
Week 2-3	Fork and validate DualMap baselines
Week 2-4	Implement Token-Budgeting + Start IEEE report draft
Week 3-5	Run Effective Request Capacity simulations and generate plots + Finish IEEE report drafting
Week 5-6	Finish IEEE report + submit

X. Reproducible Code and Artifacts

A. Code Repository

All code, data, and reproduction scripts are submitted on D2L:

The submission includes:

- `phase_aware_lwl_global_scheduler.py` — Idea1 implementation (Unified Phase-Aware LWL)
- `adaptive_k_escape_global_scheduler.py` — Idea2 implementation (Adaptive-k Escape)
- `evaluation/paper_figures_7b/` — All data, scripts, and output figures
- `scripts/reproduce_paper_7b.py` — Main evaluation script

B. System Requirements

- Python 3.10+
- 8 NVIDIA GPUs (e.g., A100, H100) with 40 GB+ VRAM each, or equivalent
- CUDA 12.0+, cuDNN 8.0+
- 512 GB total system DRAM (for vLLM KV cache buffers)
- vLLM v0.4.2 (`pip install vllm==0.4.2`)
- DualMap simulator dependencies (see requirements.txt)

C. Step-by-Step Reproduction Instructions

1) Step 1: Environment Setup:

```
# Download the repo and open a IDE of choice
(We used Visual Studio Code)

# Create Python virtual environment
python3.10 -m venv venv
source venv/bin/activate

# Install dependencies
pip install -r requirements.txt
pip install -r DualMap_paper7B_with_\
Idea1_Idea2/requirements.txt

# Verify vLLM installation
python -c "import vllm \
print(vllm.__version__)"
```

2) Step 2: Download Workload Traces:

```
# Download Mooncake open-source traces \
https://github.com/kvcache-ai/ \
Mooncake/blob/main/FAST25-release/traces/
```

3) Step 3: Start vLLM Replica Instances:

```
# Launch 8 vLLM replica instances
(each on a dedicated GPU)
# This opens ports 8081-8088 for
HTTP API access
bash scripts/run_instances_7b_template.sh
```

4) Step 4: Run Baseline Reproduction (Optional Validation):

```
# Reproduce the DualMap paper baseline
results
python scripts/reproduce_paper_7b.py \
  --scheduler dualmap \
  --output-dir evaluation/baseline_results/ \
  --workload conversation toolagent

# Plots will be saved to:
# evaluation/baseline_results/figures/
```

5) Step 5: Run Phase-Aware LWL (Idea1) Evaluation:

```
# Run the proposed Phase-Aware LWL scheduler
# Evaluates SLO Attainment, P50/P90 TTFT, \
Cache Hit Rate, etc.
python scripts/reproduce_paper_7b.py \
  --scheduler phase_aware_lwl \
  --output-dir evaluation/idea1_results/ \
  --workload conversation toolagent \
  --request-rates 1 2 3 4 5
```

6) Step 6: Run Adaptive-k Escape (Idea2) Evaluation:

```
# Run the Adaptive-k Escape variant
python scripts/reproduce_paper_7b.py \
  --scheduler adaptive_k_escape \
  --search-k 8 \
  --output-dir evaluation/idea2_results/ \
  --workload conversation toolagent \
  --request-rates 1 2 3 4 5
```

7) Step 7: Generate Comparison Figures:

```
cd evaluation/paper_figures_7b/scripts/
python plot_figure3_with_ideas.py \
  --baseline ../../../
  evaluation/baseline_results/
  --idea1 ../../../ \
  evaluation/idea1_results/
  --idea2 ../../../ \
  evaluation/idea2_results/
  --output-dir ../outputs/

# Repeat for other figures
python plot_figure4_with_ideas.py ...
python plot_figure10_with_ideas.py ...
python plot_figure11_with_ideas.py ...

cd ../../..
```

8) Step 8: Generate Final Tables:

```
python evaluation/paper_figures_7b/ \
scripts/_common.py
  --input evaluation/idea1_results/ \
  --format latex \
  --output final_tables.tex
```

D. Expected Runtime and Output

Total Reproduction Time: Heavily depends on compute hardware. May range from few hours to few days. In our case, it took around 4 days to run as our compute (lab server) was shared with other students at Michigan State.

Outputs:

- `evaluation/idea1_results/metrics.json` — Raw metrics for all request rates and workloads
- 8 PNG figures (SLO, TTFT, E2E, Cache Hit, Load Balance)

E. Validation Checklist

After running the reproduction, verify:

1) Baseline reproduction matches DualMap paper (Figure 3–4) within $\pm 2\%$
2) Phase-Aware LWL achieves $\geq 87\%$ SLO Attainment at req_rate=5 Conversation
3) Phase-Aware LWL P90 TTFT at req_rate=5 is ≤ 4.0 seconds (vs. DualMap 4.4s)
4) All 8 output figures contain data for both workloads (Conversation + Tool&Agent)
5) LaTeX tables compile without errors (verify `pdflatex final_tables.tex`)

F. Troubleshooting

- **vLLM fails to start:** Ensure GPUs are available (`nvidia-smi`). Reduce batch size in `run_instances_7b_template.sh` if OOM errors occur or use smaller model.
- **Simulator times out:** Increase heartbeat interval in `scripts/reproduce_paper_7b.py` from 100ms to 200ms.
- **Figure generation fails:** Ensure all data CSV files exist in `evaluation/idea1_results/`. Check that Matplotlib is installed (`pip install matplotlib`).

REFERENCES

[1] L. Zheng et al., "SGLANG: Efficient execution of structured language model programs," arXiv.org, Dec. 12, 2024. https://arxiv.org/abs/2312.07104v2.

[2] V. Srivatsa, Z. He, R. Abhyankar, D. Li, and Y. Zhang, "Preble: Efficient distributed prompt scheduling for LLM serving," arXiv.org, May 08, 2024. https://arxiv.org/abs/2407.00023v2.

[3] R. Qin et al., "Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot," 2025. https://www.usenix.org/conference/fast25/presentation/qin.

[4] Y. Zhong et al., "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," 2024. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

[5] Y. Yuan, P. Zuo, B. Wang, Z. Chen, Z. Tan, and Z. Yu, "DualMap: Enabling both cache affinity and load balancing for distributed LLM serving," arXiv.org, Feb. 06, 2026. https://arxiv.org/abs/2602.06502v1.

[6] G.-I. Yu et al., "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538.

[7] W. Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 611–626.

[8] J. Yao et al., "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion," in *Twentieth European Conference on Computer Systems (EuroSys '25)*, Rotterdam, Netherlands: ACM, Mar. 2025.

[9] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[10] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve," in *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[11] P. Patel, E. Choukse, C. Zhang, A. Shah, I. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient Generative LLM Inference Using Phase Splitting," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.