

KVCompass: A Discovery Plane for Internet-Scale Prefix Cache Sharing in LLM Serving

SHIYI LING, University of California, Santa Cruz, USA

FEI FANG, University of California, Santa Cruz, USA

CHEN QIAN, University of California, Santa Cruz, USA

KV cache reuse is one of the most effective ways to reduce LLM inference cost, but today its benefits are largely confined to a single cluster. When a provider serves the same model across geographically distributed regions, each region may repeatedly prefill the same popular prefixes, wasting GPU time and increasing time-to-first-token (TTFT). Exploiting reuse across regions involves discovering which worker currently holds a reusable KV cache for each request's prefix. We show that this KV cache discovery problem is structurally harder than conventional object lookup. Unlike hash-placement mechanisms such as DHTs, where an object's location is a deterministic function of its key, KV caches are created on the workers that serve requests, and their holder sets change with subsequent requests and KV cache evictions. KV cache discovery therefore calls for decoupling KV cache identity from placement. We present KVCompass, a discovery plane for Internet-scale KV cache sharing. The discovery index supports variable-length prefix lookup with constant-time local table reads at the edge, allowing each request to be forwarded to the worker that best balances KV cache reuse against load and network distance. Evaluation includes a 64-worker geo-distributed simulation and a 14-worker three-continent real-machine testbed. Evaluation shows that KVCompass achieves 1.3–14× lower P99 TTFT than the baselines, up to 3.7× lower P99 end-to-end latency, up to 4.7× the throughput, and a peak load factor within 2% of the Oracle floor.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: Large Language Models; KV cache prefix caching; Geo-distributed systems

ACM Reference Format:

Shiyi Ling, Fei Fang, and Chen Qian. 2026. KVCompass: A Discovery Plane for Internet-Scale Prefix Cache Sharing in LLM Serving. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 3, Article 68 (December 2026), 26 pages. <https://doi.org/10.1145/3857371>

1 Introduction

Large language models (LLMs) have rapidly become a core building block of modern computing, supporting a wide range of applications like interactive assistants, code generation, and document analysis [21, 27, 56, 58]. As LLM adoption expands, inference workloads are growing in both scale and diversity. Processing a single request can cost orders of magnitude more than a traditional query [3, 7, 38], and cloud providers face increasing pressure to reduce cost while meeting strict latency requirements such as time-to-first-token (TTFT) [36, 42], making efficient LLM serving a first-order systems challenge [21, 36, 51].

In practice, many LLM requests share common prefixes, such as system prompts, retrieved documents, or conversation histories [22]. A large-scale measurement study at a major cloud

Authors' Contact Information: Shiyi Ling, sling9@ucsc.edu, University of California, Santa Cruz, Santa Cruz, California, USA; Fei Fang, ffang6@ucsc.edu, University of California, Santa Cruz, Santa Cruz, California, USA; Chen Qian, cqian12@ucsc.edu, University of California, Santa Cruz, Santa Cruz, California, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2476-1249/2026/12-ART68

<https://doi.org/10.1145/3857371>

provider [45] found that 10% of KV cache blocks account for 77% of all reuses, and that prefix-cache hit rates reach 54–62% across representative traces. A widely adopted technique to exploit this structure to improve LLM serving efficiency is KV cache reuse [11, 21, 28, 36, 58]: when multiple requests share a common prefix, the computed KV cache can be shared among requests, avoiding expensive prefill computation and reducing latency and serving cost. Commercial prefix caching APIs already exploit this within a single cluster, reducing cost by 50–90% for long prompts [1, 33].

However, these gains remain confined to a single cluster, while real-world LLM serving is fundamentally a global, cross-region problem [6, 17]. The demand for LLM serving is geographically dispersed: in WildChat [55], a corpus of nearly one million ChatGPT interactions, requests span 201 countries and 178,724 users. At Internet scale, as major inference providers increasingly deploy the same model across tens of geographic regions to serve global users with low latency [9, 35], each region independently prefills the same popular prefixes, wasting GPU compute at scale. When we replay WildChat [55] in its arrival order, Figure 2 shows that 6.6% – 21.2% reusable prefixes exist *only* in other regions at arrival time. However, these reuse opportunities are invisible to any KV cache confined to a single cluster or region, no matter how well it is managed. Harvesting them requires extending KV cache reuse to Internet scale, which in turn requires answering a new question for every request: *which worker currently holds a reusable KV cache for this prompt’s prefix?* This is the **KV cache discovery** problem, a challenge that is fundamental to Internet-scale LLM serving and unaddressed by existing systems.

KV cache discovery at Internet scale is harder than it appears. A centralized directory maps prefix hashes to KV cache locations, but each query pays a wide-area round trip that adds directly to TTFT. The natural alternative is to turn to distributed hash-based schemes such as consistent hashing [20] or distributed hash tables (DHTs) [30, 37, 39, 43]. However, DHTs share a **fundamental limitation**: an item’s placement is determined by its content hash rather than by where it was created [29, 46], which breaks down for KV caches: a cache entry materializes on whichever node happened to serve the prior request, not on a node designated by a hash. Forcing KV caches onto hash-designated nodes would incur migration and replication costs that negate the savings that reuse is meant to provide. KV caches thus demand **placement–identity decoupling**, which structurally rules out the entire hash-placement family, regardless of how such schemes are tuned. Therefore, no prior distributed naming or location service addresses KV cache discovery at Internet scale.

We propose a KV cache discovery plane for Internet-scale KV cache sharing. KVCompass is the **first system to apply space-efficient minimal perfect hashing (MPH) [4] to Internet-scale LLM serving**. MPH’s space efficiency allows the full KV cache index to be replicated in the DRAM of edge nodes in every region, making every lookup a local memory operation rather than a round trip to a remote coordinator. Its explicitly written values achieve placement–identity decoupling: the index tracks where caches land, with no control over where they go, so any worker can register a KV cache entry and any region can discover it with a local lookup. The prefix hash identifies the KV cache entry, while its value records the workers holding it and their storage tiers. Updating the value changes the holder set without changing the prefix identity. The index is organized as parallel tables at prefix lengths derived from the network latency from the edge to each region, so a lookup matches at the lengths where the prefill time a match saves exceeds the network latency of reaching its holder. Sharded maintenance units own all writes to the index, so the read path needs no locks and no cross-node coordination. At forwarding time, a joint score evaluates every worker rather than only the KV cache holders, combining network distance, queuing delay, cache tier, and residual prefill. Together, these properties allow KVCompass to reduce serving cost at Internet scale and provide the first principled answer to the KV cache discovery problem.

This paper makes the following contributions:

(1) We are the first to formalize the **KV cache discovery plane** as the missing layer in **Internet-scale LLM serving**, identifying its four objectives: placement–identity decoupling, geography-adaptive match granularity, local and low-overhead lookups, and load balancing, a combination that no prior distributed location service supports.

(2) We design and implement KVCompass, a prototype KV cache discovery plane built on a space-efficient hash table, with **local latency-derived multi-level prefix lookup with $O(1)$ table reads** and a **joint score** that balances KV cache reuse and worker load. Evaluation shows KVCompass reduces TTFT by 1.3–14 $\times$ and end-to-end latency by up to 3.7 $\times$, delivers up to 4.7 $\times$ higher throughput, and keeps peak load factor within 2% of the Oracle floor.

Section 2 motivates the problem, Section 3 presents the design, and Section 4 describes the implementation. Section 5 evaluates KVCompass. Sections 6, 7, and 8 discuss extensions, related work, and conclusions. Our code is available at <https://github.com/011shiyiLing/KVCompass>.

2 Background and Motivation

This section motivates an Internet-scale discovery plane for cross-region KV cache reuse. We first quantify the cross-region reuse opportunity in a production workload (Section 2.1), then derive four requirements for an effective discovery plane (Section 2.2), and finally explain why directly deriving cache locations through hash-based placement does not satisfy these requirements (Section 2.3).

2.1 Cross-region LLM Serving

An LLM serving system [21, 58] hosts model replicas on multiple GPU workers and responds to each user prompt by generating a stream of output tokens. A request is processed in two phases: a *prefill* phase that receives the prompt and computes its KV cache, followed by a *decode* phase that generates one token per step. Serving quality is measured along both the latency and throughput dimensions. Users experience time to first token (TTFT), which is largely determined by queueing and prefill, as well as end-to-end latency, which additionally includes autoregressive decoding. Providers care about the aggregate throughput sustained by the worker pool, as it directly affects the serving cost per request [21, 59].

Prompts frequently share long common prefixes, such as system prompts, few-shot templates, and multi-turn history. Therefore, modern engines retain the KV cache of served prefixes and skip prefill for the shared part, simultaneously improving TTFT and throughput [1, 33, 36, 58]. As illustrated in Figure 1, request 0 first prefills its prompt and uses the resulting KV cache for decoding. Because identical prefixes produce identical KV states, request 1 can reuse the KV states of the three shared tokens and prefill only the remaining tokens. Longer shared prefixes yield greater prefill savings.

However, these gains are confined to a single cluster within one region that holds the KV cache. Nowadays, production providers deploy serving clusters in geo-distributed regions. These clusters serve the same model independently, and each keeps its own local KV cache. The natural next step is to extend KV cache reuse to global scale. This raises a question: *do user requests in one region share prefixes already cached in another?*

To answer this question, we analyze the WildChat [55] workload. We take the first 1,024 tokens of each request as its prefix. We keep the prefixes reused by at least two distinct users, and map each request’s origin country to the nearest of 8 deployment regions. A shared prefix is *cross-region*

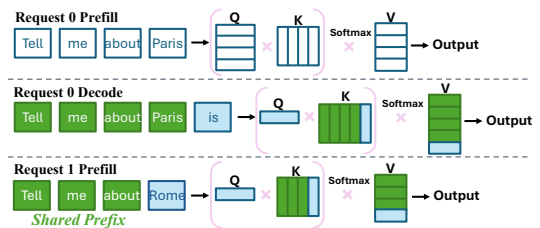


Fig. 1. KV cache reuse via prefix sharing.

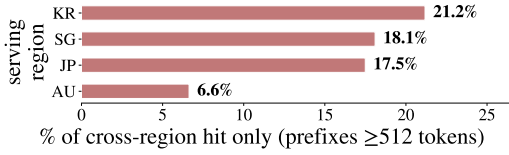


Fig. 2. Cross-region only KV cache hit of ≥ 512 -token prefixes in WildChat [55]

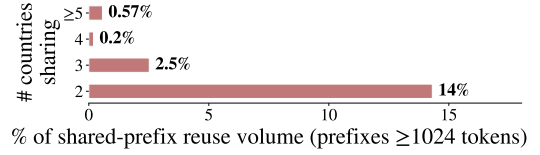


Fig. 3. Cross-region reuse in WildChat [55] for prefixes of ≥ 1024 tokens.

when its sharing users map to at least two different regions; only these reuses face a choice between recomputing the prefix locally and reusing a KV cache in a remote region. Figure 3 shows the volume of the cross-region prefixes, bucketed by how many countries share the prefix. Prefixes shared between two countries carry 14%, while those shared between three countries carry 2.5%. Overall, approximately 17% of all shared-prefix reuse volume crosses a single region boundary.

However, in today's deployments a router only knows the KV caches stored on workers in its own region, so it cannot find a remotely computed KV cache and has to recompute the prefix from scratch. This 17% of the potential reuse thus has no chance to happen. Losing this share also costs the most, since prefixes shared across regions are usually the long, popular ones whose recomputation is the most expensive. **Cross-region KV cache reuse** is therefore both large in volume and worth exploiting per request. Thus, the *KV cache discovery at Internet scale*, looking up which worker anywhere in the world already holds a reusable KV cache, becomes an important challenge. Appendix D provides additional reuse analysis on Qwen-Bailian [45] and Mooncake [36] production traces.

2.2 Objectives for a KV Cache Discovery Plane

The preceding analysis establishes a substantial opportunity for cross-region prefix KV cache reuse. Realizing this opportunity, however, requires a discovery plane that can locate reusable caches across regions without adding significant latency or creating new load imbalances.

Suppose a provider operates a serving worker pool across R geo-distributed regions. The KV cache for a prefix may be stored in a different region's worker from the request that could reuse it. Realizing cross-region reuse therefore requires an **Internet-scale KV cache discovery plane**. For each incoming request, this layer identifies workers that currently hold reusable prefix KV caches and combines those matches with network and load information to select the locally estimated fastest serving worker. To discover reusable KV caches across regions correctly and efficiently, KVCompass needs to satisfy four objectives.

(O1) KV caches need to be discoverable wherever the requests that compute them land, not where their identities hash to. In conventional location services using hash-based placement, an item lives where its identity dictates: a content hash maps the item to a designated node, and a lookup recomputes that mapping to find it. However, a KV cache entry obeys no such rule. It is not a deliberately placed object, but a by-product of computation that materializes on whichever worker happened to serve the request that produced it. Therefore, a KV cache discovery plane cannot derive a KV cache's location from its prefix's hash value; it needs to locate KV caches wherever they have been served, with no control over where they go. We call this requirement *placement-identity decoupling*.

(O2) Prefix match lengths need to adapt to both the request content and geography. A prompt of a request does not share a fixed number of tokens with the KV caches stored on different workers. For example, a 6000-token prompt may share its first 256 tokens with some cached entries,

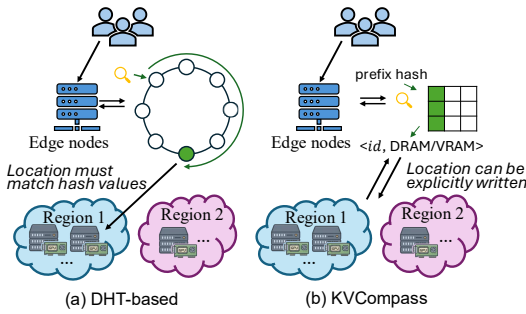


Fig. 4. Comparison of hash-placement and explicit-write approaches for KV cache discovery.

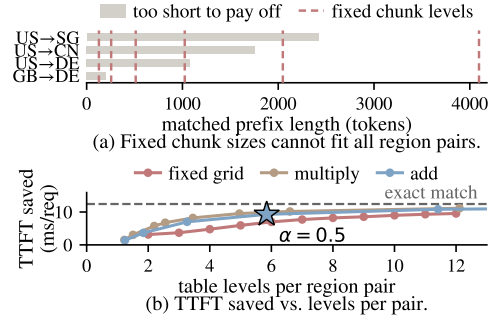


Fig. 5. Network latency-derived index levels versus a fixed chunk grid.

its first 2048 with others, and its first 5120 with a third set [25]. Therefore, looking up every candidate at one fixed length considers only a fraction of the matches longer than that length, and misses the matches shorter than the length entirely. Extending the index to a fixed length of multiple chunk sizes does not resolve this problem either, because at Internet scale the length worth matching is not a property of the prompt itself alone. Reusing a KV cache in a remote region pays off only when the prefill time it saves exceeds the network latency of reaching the worker holding the KV cache, and that latency depends on which region the request is forwarded to. As Figure 5(a) shows, the shortest prefix length worth forwarding ranges from a few hundred to a few thousand tokens, depending on the distance between regions. Suppose every region used one fixed grid of lengths. For any given region, many of the grid's lengths then fall below its shortest worthwhile length toward some remote region, and a match found at those lengths saves less prefill time than the forwarding latency it pays, so these are matches we do not want to discover. Thus, the discovery plane needs to judge each candidate at match lengths that reflect both the prompt's sharing structure and the geographic distance between the user and the KV cache holder.

(O3) Lookups need to be local and effectively free. A lookup that triggers a network round trip to a remote node adds directly to TTFT, eliminating the time savings that KV cache reuse is meant to provide. Thus, every lookup has to resolve locally at the edge node, without off-node coordination. At Internet scale this imposes two demands. First, the state behind a lookup needs to be small enough to reside at every edge node rather than at a centralized directory. Second, each lookup needs to be fast. Because an edge node performs a lookup for every request at tens of candidate prefix lengths, the index answers tens of queries for every request served. Multiplied across all requests, even a small cost per lookup becomes the dominant work on the edge, and an overloaded edge stalls every request behind it. O2 sharpens this objective further. Because the worthwhile match length varies with the geographic location of each holder, a lookup cannot settle for a single fixed depth, yet it has to check all relevant lengths within this same tight local budget.

(O4) Discovery needs to avoid creating hotspots. Popular prefixes are the most valuable to discover, and they also attract the most requests. But forwarding every matching request to the few workers that already hold a popular prefix's KV cache will overload those workers while the rest of the workers sit idle. As requests concentrate on one worker, its queueing delay grows and eventually outweighs the prefill time saved by reusing the KV cache. Therefore, the discovery plane needs to weigh this saving against each worker's load and network distance, and may prefer an idle worker that recomputes the prefix over a congested or distant one.

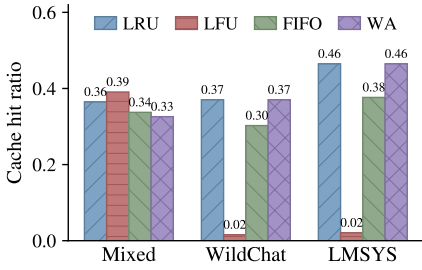


Fig. 6. KV cache hit ratios by eviction policy.

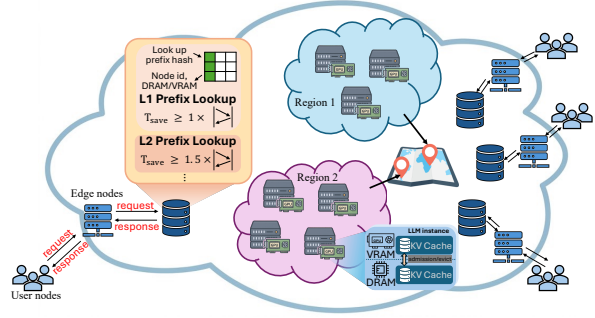


Fig. 7. KVCompass overview.

2.3 Why Hash-Directed Cache Placement Is Insufficient

A natural idea for a location service in an Internet-scale system is a DHT such as Kademlia [30] or Chord [43], a distributed implementation of consistent hashing used within clusters for prefix KV cache affinity [36]. However, they all assume *placement equals identity*: an item's location is determined by its hash value, as Figure 4 (a) illustrates, the assumption that O1 rules out. To mediate a KV cache's created and hash-designated locations, a hash-placement scheme would have to migrate each entry to its hash-designated node, which costs hundreds of megabytes of wide-area bandwidth per entry. In the other direction, the scheme would index only the entries that already sit at their hash-designated nodes; since KV caches are rarely at these nodes, most KV caches would become undiscoverable. Therefore, the entire hash-placement family, including Chord, Kademlia, consistent hashing, and Rendezvous hashing, is invalid as a foundation for this discovery plane.

The KV cache discovery plane instead needs a data structure whose value field is *explicitly written* by whoever holds the KV cache, not derived from the key, as Figure 4 (b) illustrates. Hashing may decide where a mapping is stored *inside the index*, but never where the KV cache itself lives. Such an explicit-write structure satisfies placement–identity decoupling (O1) by construction. Section 3 shows how KVCompass builds on this principle to meet the remaining objectives.

3 Design of KVCompass

3.1 System Overview

We target an LLM serving deployment that spans *multiple regions*. Regions are geographically distributed, in different countries or even continents. Each region hosts a set of *workers*. A worker is one inference instance, holding the full model replica on one or more GPUs, and it keeps the KV caches of the prefixes it has served in its VRAM and host DRAM. A worker reports a *KV cache event* whenever its local KV cache changes state, admitting a newly computed prefix, moving one between VRAM and DRAM, or evicting one. Workers within a region share a high-speed datacenter network, while traffic between regions crosses the wide-area network, with round-trip times orders of magnitude larger. Each region also runs several *edge nodes*, CPU-only gateways separate from the workers. A user request arrives at its nearest edge node, and is forwarded to a worker, in the same region or in a remote region.

KVCompass is a KV cache discovery plane for this distributed LLM serving. Forwarding every request to the few workers holding popular prefixes would create the hotspots that (O4) warns against, so KVCompass balances KV cache affinity and worker load jointly. For each incoming request, it identifies which workers across all regions hold a reusable KV cache for the request's

Table 1. Terms and notations.

Term / Notation	Meaning	Example values
worker	inference instance holding KV caches	$\sim 10-10^2$
region	geographic site hosting a set of workers	~ 10
edge node	gateway of a region, hosting a lookup unit	several per region
lookup unit	keyless view of the discovery index	one per edge node
shard	a logical group of the discovery index entries, assigned by hashing the key;	10^4-10^6 entries per shard
maintenance unit	applies KV cache events to its shard and pushes the updates to the lookup units	one per shard
L	the number of the discovery index levels	~ 6 per edge node
E	the total number of the discovery index entries	10^5-10^7
n_1	first latency-derived prefix length	10^2-10^3 tokens
$\text{WAN}(o, r)$	one-way latency from region o to worker r 's region	$\sim 10^2$ ms
β_r	worker r 's prefill rate	~ 0.1 ms/token

prefix, then weighs the saved prefill cost against each candidate's load and geographic network latency, and forwards the request to the worker with the lowest expected completion time.

KVCompass is built from a single logical data structure and two kinds of software components that act on it. The data structure is the *multi-level ludo discovery index* (Section 3.2), which maps prefix hashes to the workers currently holding the corresponding KV cache. *Maintenance units* are the sole writers of the index and apply KV cache events to it. *Lookup units* answer the discovery query of every request: which workers hold a reusable KV cache for its prefix, and in which storage tier; each holds a local view of the discovery index, a compact slot-only copy without the keys (Section 3.3). The components act on the discovery index through four operations, lookup for forwarding the request and insert, update, and remove for the KV cache events (Section 3.4). We also analyze the memory footprint, operation costs, and control traffic of KVCompass in Section 3.5. Table 1 summarizes these terms and the notations used in the rest of the paper.

Figure 7 shows the request and update flows. At the nearest edge node, the lookup unit queries latency-derived prefix lengths and returns matching workers and their KV cache tiers. KVCompass scores all workers across regions using residual prefill time, load, and network distance, then forwards the request to the winner. Workers report KV cache admissions, tier changes, and evictions to maintenance units, which update the discovery index and lookup units.

KVCompass does not control when a worker admits, demotes, or evicts a KV cache. Figure 6 compares four eviction policies on three workloads. LFU achieves the highest KV cache hit ratio on Mixed, while LRU performs best on WildChat and LMSYS. With no policy best across workloads, KVCompass leaves the choice to each worker and only tracks the resulting KV cache events.

3.2 The Multi-Level Ludo Discovery Index

The discovery index is the shared state of KVCompass: lookup units read it and maintenance units write it. We first recall the hash structure it builds on and why it is suitable for our problem, then describe the modifications that the KV cache discovery plane demands of the discovery index.

The index builds on Ludo hashing [40], a key-value lookup engine that combines a minimal perfect hash function (MPH) [4] with cuckoo displacement: given any set of key-value mappings, it constructs a compact lookup structure that returns a key's value in $O(1)$ with a few memory accesses, without storing the keys themselves, cutting space by up to 90% compared with state-of-the-art hash tables. The Ludo table is sized at build time for its expected entry count and does

not grow in place. Its use of hashing is fundamentally different from the hash-placement family dismissed in Section 2.3: Ludo hashes a key only to locate the table slot holding a mapping that the writer chose, so the value can name any node, and two workers that independently cache the same prefix register their own locations under the same key, with no migration and no structural conflict. The explicitly written value yields the placement–identity decoupling that (O1) calls for, letting the plane track KV caches wherever computation leaves them. The keyless compactness makes parallel tables at many match lengths affordable, so multi-granularity matching (O2) need not multiply memory cost, and, combined with $O(1)$ lookup, lets the discovery index replicate in the local memory of every edge node, keeping lookups local and effectively free (O3).

However, a flat Ludo table does not yet satisfy these objectives. As a key-value design it maps each key to a single value, hence exactly one worker, while in Internet-scale sharing the same prefix is often cached by many workers at once. It operates at one fixed prefix length, treating a nearby holder and a distant one equally, and because it stores no keys, a lookup for a prefix that was never inserted can return a false hit. Therefore, KVCompass redesigns the structure in three ways, which are described below.

Tracking dynamic, multi-holder KV cache sets. The same prefix is cached by whichever workers happened to serve requests for it; the holder set churns continuously as KV caches are admitted under load and evicted under memory pressure, and each holder independently moves the prefix between VRAM and host DRAM, so a discovery index needs to resolve to a mutable set of holders and their storage tiers under a high rate of updates. KVCompass organizes the index as Ludo tables. Each entry indexes one prefix at one length within its table. For example, the entry indexing the first c tokens of a prompt is keyed by a 63-bit key, the truncated SHA-256 digest of those c tokens; c here is drawn from the network latency-derived set of indexed lengths constructed below, so entries at many different lengths coexist in the same table. Each entry stores a packed value with one 2-bit tier field per worker in its table. The field is 01 for VRAM, 10 for DRAM, and 00 otherwise. Each region has its own table. A 32-bit value holds tier fields for up to 16 workers, and a 64-bit value holds up to 32. Each field uses 2 bits. Every edge node keeps all region tables and combines their lookup results to find the holders. A region with more than 32 workers requires additional tables. When an entry is created, all worker fields in its table are initialized to 00. A worker’s field becomes 01 or 10 when it caches the prefix. One read returns all worker fields in that table, and a tier change is an in-place 2-bit write. We exclude SSD because fetching a prefix on request arrival adds an SSD read to the serving path, typically costing more than recomputing it.

Matching prefixes at distance-aware granularities. The discovery index needs to answer one question after finding the matched prefix: is this prefix match long enough to be worth forwarding to that region to reuse? Fixed chunk sizes of prefix cannot answer it (O2). Forwarding to another region to serve the request adds the round trip from the edge to the target region, while the match buys skipped prefill time, which grows linearly with the matched tokens at the per-worker prefill rate β in ms/token. A match of n tokens saves about βn ms, so this forwarding decision pays off once this KV cache reuse saving covers the added round trip. Therefore, KVCompass cuts the first indexed length at exactly this point. This shortest profitable prefix length is the first *latency-derived prefix length* n_1 , and it scales with the measured latency from the edge to that region. Within the edge’s own region the round trip is so small that this formula would put n_1 at a few dozen tokens, guarding a cost that hardly exists, so same-region matching starts at a short fixed minimum.

But one level per region is still not enough. The first latency-derived prefix length only tells the edge whether that forwarding is worthwhile; to pick among candidates, the forwarding decision needs to know how much each match saves. With a single indexed length a very long token match and a barely profitable one would look identical. KVCompass thus indexes several levels of increasing lengths above each region’s n_1 , with the same number of levels toward every region.

Figure 5(b) compares different choices of the level lengths over the WildChat [55] reuse distribution. Latency-derived levels beat latency-blind fixed grids at every table budget, and lengths spaced at equal steps of half of n_1 , giving n_1 , $1.5 n_1$, $2 n_1$, and so on, capture about three quarters of the exact-match upper bound with about six levels. KVCompass adopts these steps. We also measured multiplicative growth, but a match falls to the nearest indexed length below it, and multiplicative spacing grows coarsest exactly where matches are longest and most valuable, whereas the equal-step spacing keeps the under-crediting bounded by half of n_1 at every match length. The same-region lengths instead double from the short minimum, following the chunk sizes by which workers organize their KV caches. All the level lengths derive from a stable WAN latency matrix and per-worker prefill rates, so the whole structure can be computed offline and adds no extra overhead to the request serving path.

Suppressing false hits without inflating the discovery index size. Because Ludo does not store the keys it indexes, a lookup for a prefix that was never inserted may collide with an unrelated entry and name a worker that does not actually hold it, so the request is forwarded to a worker without the prefix and pays a needless WAN round trip. Storing full keys at every edge node would eliminate these false hits but inflate the discovery index far beyond DRAM budgets, defeating the space efficiency that makes edge local lookup possible. KVCompass instead lets a false hit surface and absorbs it cheaply: the worker a request is forwarded to serves only the prefix it truly holds and recomputes the rest, so the answer is always correct and the only cost is one forwarding that saves less prefill than expected. To keep such forwardings rare, each edge node discounts a worker's claimed match by a *cache belief* maintained from past responses, so a worker that often does not actually hold what it claims stops attracting requests, while the maintenance units keep the full keys off the forwarding path and use them only for collision detection when updating the index.

3.3 System Components

There are two kinds of software components that act on the discovery index. Lookup units resolve which workers hold how much of a prompt's prefix at the edge node. Maintenance units turn KV cache events into the discovery index writes and propagate them to the lookup units.

Lookup units. A lookup unit runs on the edge node and holds a view of every shard. Each lookup unit queries the multi-level tables at multiple lengths, derived from its measured network latencies toward each region. The lookup view stored on every edge node is identical. A worker registers each cached prefix at the union of lengths that any edge may query, so only the queried lengths differ across regions. Identical views keep the edges interchangeable. A new or recovering edge can bootstrap from the maintenance units or copy any other edge, and if every edge node of a region fails, another region's edge can stand in. Holding everything at every edge node also costs little, since a view does not store the keys. Answering a query needs only the slot states, so the keys stay with the maintenance units.

When a request arrives, KVCompass computes the prefix digests at the edge's query lengths, reads the $\langle \text{worker}, \text{tier} \rangle$ holders at each length from the lookup unit's view, evaluates the joint score over all workers across regions, and forwards the request to the winner. Queries are plain memory reads with no locking, because each shard has a single writer and its update messages are applied in the order they were sent.

Maintenance units. Maintenance units construct and update the discovery index and are its sole writers. Each unit owns one shard, the keys whose $\text{hash}(K) \bmod N$ matches its identifier, and runs as a CPU-resident process co-located with a worker. Shards split the key space among the N writers, while views replicate the whole key space at every edge. Each maintenance unit therefore sends its shard's updates to every edge when KV cache events happen, and each edge receives the update information of all N units. On each KV cache event, the unit locates the key in its shard,

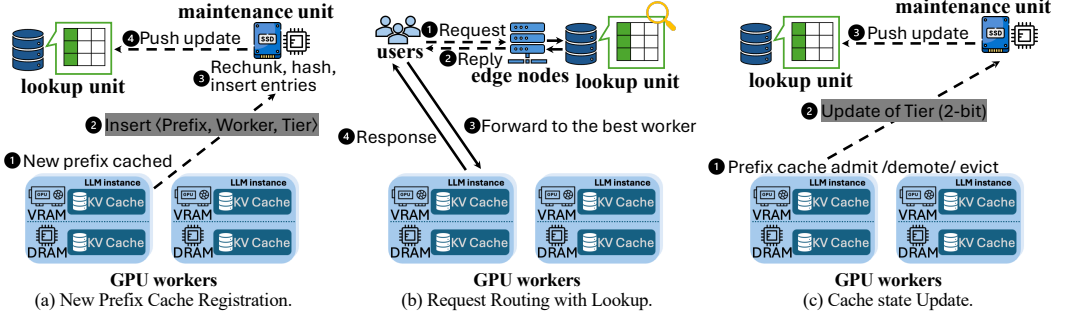


Fig. 8. Discovery plane operations. (a) A worker registers a newly cached prefix. (b) The edge node looks up a request and forwards it to the best worker under its local view, as determined by the joint score. (c) A KV cache state change updates the worker's 2-bit storage-tier field.

applies the implied 2-bit field edit, and sends the update to the lookup unit, so the lookup views stay in sync without communicating with each other.

3.4 Operations of the Discovery Index

KVCompass exposes four operations on the discovery index: lookup when a user sends an inference request at an edge node; insert, update, and remove when a worker's KV cache changes state. Figure 8 illustrates the flow of each, and we describe them in turn.

Lookup. As Figure 8(b) shows, a user request lands at the edge node closest to its origin region o ①, the lookup unit answers with the matching workers and their tiers ②, and the request is forwarded to the best worker ③, which will return the response to the user ④. To find the matches, KVCompass scans the discovery indexes on the Ludo table at every level l whose indexed length c_l fits the prompt, obtaining for each worker r its longest matching prefix length m_r and storage tier t_r . KVCompass ranks candidates with a joint score that weighs each worker's load, KV cache affinity, and network and reload cost in one formula. The score is evaluated over all workers across regions rather than only the KV cache holders. Under load, an idle worker that recomputes the prefix from scratch frequently beats a distant or congested holder, the preference that (O4) calls for. The discovery plane forwards the prompt to the worker minimizing $\text{score}(r) = \text{WAN}(o, r) + \widehat{T}_q(r) + (1 + \lambda)\beta_r(L_{\text{prompt}} - b_r m_r) + \gamma b_r m_r \mathbf{1}[t_r = \text{DRAM}]$, where $\widehat{T}_q(r)$ is the estimated queueing delay of r , β_r is the per-worker prefill rate in ms/token, and γ is the per-token PCIe reload cost, charged only when the matched prefix is DRAM-resident. Worker r 's cache belief $b_r \in [0, 1]$ is the exponentially weighted moving average of the ratio of actually reused prefix tokens, reported in each response, to the match claimed by the edge. Frequent false claims lower b_r and the effective match $b_r m_r$, making the worker less likely to win the joint score.

The score consults $\widehat{T}_q(r)$ for workers in every region, yet an edge observes only its own region directly. Each edge node therefore keeps a per-worker queue estimate, read live for the local region and refreshed for remote regions through two channels. A cross-region response carries the serving region's per-worker queue vector, which the requesting edge folds into its estimate with an exponentially weighted moving average of weight w . Each region also broadcasts its vector at a fixed period Δ , and the broadcast is suppressed toward any region that recently received a piggybacked report. Staleness is thus bounded on both paths, within one response interarrival for a busy pair and one broadcast period for an idle one. Freshness also matches demand, since the pairs whose forwarding depends most on accurate estimates are exactly the pairs refreshed most

often. The queue vector adds only a few dozen bytes to responses already crossing the WAN, so the fresher estimates cost no additional control traffic.

The factor $(1 + \lambda)$ prices recomputation by spare capacity. With $\lambda = \min(\lambda_{\max}, \lambda_q)$, where $\lambda_q = \min_{r'} \widehat{T}_q(r')/\tau$ for a normalization constant τ , the capacity price stays near zero while idle workers exist and grows as the pool saturates. On an idle worker, recomputation costs only the request's own prefill time and often beats a WAN round trip to a distant holder. At saturation every recomputed token delays queued requests, so reuse saves aggregate work and the inflated multiplier shifts the choice toward KV cache holders. Each edge computes λ from its own estimates, so the pricing reflects the spare capacity that particular edge can see when it forwards. A stale estimate can misprice λ , but the error is bounded, since λ is capped at $\lambda_{\max}$ and $1/\tau$ -Lipschitz in the estimates, and it skews at most that one forwarding decision.

Insert. As Figure 8(a) shows, when a worker completes a new prefill, the prompt's prefix is now in its VRAM ❶, and the worker registers this new copy with the discovery plane. The worker reports one registration event carrying the prefix, its own identifier, and the tier ❷. The maintenance units rechunk the prefix at every indexed length $c_l \leq L_{\text{prompt}}$ and hash each chunk into its key K_l , and the maintenance unit owning K_l inserts the entry by rewriting the worker's 2-bit slot to 01 ❸. The corresponding update message is propagated to the lookup unit ❹: a short cuckoo-move path if the key was previously absent, or a single in-place slot patch otherwise.

Request serving is not the only trigger of inserts. Once a worker serves a prefix, the resulting insert makes that copy discoverable, so popular prefixes spread across the worker pool as a byproduct of serving. However, this passive spread fails when a prefix turns hot in a region that has never served it, where every early request must either cross the WAN to a remote holder or recompute from scratch. KVCompass covers this transient by migrating a popular KV cache replica proactively, and the migration triggers an insert from the destination worker. KVCompass tracks the popularity of prefixes locally at each edge node, counting per *prefix family*, all the prefixes sharing the same first-chunk key. A prefix family turns hot once its local misses reach H within a sliding time window T_h . What migrates is the triggering request's own prefix, which contains the shared ancestor, so every sibling in the same prefix family benefits at least up to the shared portion. The migration is triggered when the region has no local holder and a remote holder's match exceeds the latency-derived prefix length of that holder's region, or when every local holder keeps losing the joint score to load. To stay off the request serving path, the replica goes only to a persistently idle worker, one whose estimated queue has stayed at zero, and it lands in the destination worker's DRAM so as not to evict the KV caches already serving from VRAM. Within a region the replica is copied worker-to-worker connected by a high-bandwidth network. Across regions moving KV tensors over the WAN is uneconomical, so KVCompass ships the prompt and recomputes the KV cache at the destination worker, and since the destination worker is idle, the recomputation takes no GPU time from serving.

Update. As Figure 8(c) shows, when a worker demotes a prefix from VRAM to host DRAM or promotes it back, it reports a KV cache event for the prefix ❶. The owning maintenance units rewrite that worker's 2-bit field at every level's discovery index ❷ and propagate the update to the lookup unit ❸. The prefix is already resident from its earlier insertion, but a cuckoo displacement may have moved it since, so the unit searches both candidate buckets for the current slot before writing the new tier code. No cuckoo move is involved, because an update changes only the stored value and relocation runs only when an insertion hits a full bucket. A single resulting (bucket, slot, value) triple is propagated to the lookup unit.

Remove. When a worker fully evicts a prefix, the maintenance unit clears its slot to 00 along the same path as in Figure 8(c) ❷. If other workers still hold the prefix, the entry remains; once the

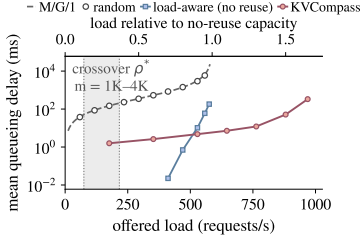


Fig. 9. Simulated mean queueing delay versus offered load (Mixed).

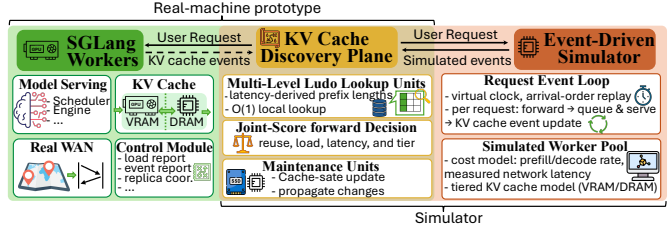


Fig. 10. KVCompass implementation. The prototype and event-driven simulator share the same KV cache discovery plane.

packed value drops to zero, the entry is removed from the table. No tombstone is required: each shard has a single writer, and stale retransmissions are rejected by the per-rank sequence check.

3.5 Performance Modeling and Analysis

We analyze discovery index memory, operation costs, and control traffic. E is the total entry count across all tables, W the worker count, R the region count, L the number of indexed levels, and β the prefill time per token in ms.

Memory footprint. Following the Ludo analysis [40], at 95% occupancy a table requires about 0.263 four-slot buckets and 2.33 classifier bits per entry. With 32-bit values, the lookup view costs $2.33 + 0.263 \times (5 + 4 \times 32) \approx 37$ bits per entry. Retaining 63-bit keys raises the maintenance cost to about 104 bits per entry. With 64-bit values, these costs rise to about 71 and 137 bits per entry. Total memory sums these costs over all table entries, with additional allocation overhead. As workers and regions are added, more tables may be needed, and prefixes held in multiple tables occupy separate entries. Each edge stores all tables, so more edges increase total lookup-view memory.

Lookup cost. A lookup makes at most L table reads per region, one for each indexed length that fits the prompt. Each read takes about three memory accesses, two for the classifier and one for the bucket, independent of E . Across R regions, table reads cost $O(LR)$ and scoring costs $O(W)$, giving $O(LR + W)$ per request. All reads are local; only request forwarding crosses the WAN.

Queueing delay under load. A queueing approximation explains when load outweighs KV cache reuse in the joint score. Model a worker as an M/G/1 queue with Poisson arrival rate a after forwarding and service time S drawn from the request mix. The Pollaczek–Khinchine formula [15] gives $\mathbb{E}[T_q] = a \mathbb{E}[S^2] / (2(1 - \rho))$ at utilization $\rho = a \mathbb{E}[S]$. A KV cache match of m tokens saves βm of prefill time. Equating these costs gives the crossover utilization ρ^* above which an idle worker that recomputes can outperform a loaded holder, motivating the score’s capacity price λ .

Figure 9 sweeps offered load in simulation on Mixed. Random forwarding follows the model despite a service-time C_v^2 of 5.4, supporting the approximation. The predicted crossover is $\rho^* = 0.13$ to 0.37 for matches of 1K to 4K tokens, suggesting that load matters well below saturation. Let λ_0 denote the no-reuse capacity, the arrival rate at which offered work equals the pool’s service rate with every prefix recomputed in full. Forwarding to the least loaded worker removes nearly all queueing below this point, with no measurable waits below $0.5 \lambda_0$, but its knee remains at λ_0 . The full score reduces mean residual prefill by 42% and shifts the knee to about $1.65 \lambda_0$, consistent with $1/(1 - 0.42) \approx 1.7$. Load awareness reduces queueing, while KV cache reuse reduces work and increases capacity.

Index write cost. Each KV cache state event edits at most L entries, one per indexed length covered by the prefix. Each edit takes amortized $O(1)$ work at its maintenance unit and produces a

(bucket, slot, value) triple with a 32-bit or 64-bit value, or a short cuckoo-move path for a new key. Propagation therefore sends at most L updates per event to each edge, independent of E .

Migration cost. A KV cache replica can be produced by transferring its tensors over the network or shipping the prompt and recomputing them at the destination. The comparison weighs network transfer time against the GPU time needed to recreate the same prefix. With s KV bytes per token and prefill time β ms/token, the model favors tensor transfer when bandwidth in bytes/ms exceeds s/β . Regional networks provide tens to hundreds of Gbps and typically exceed this threshold, while inter-region links over the public Internet fall below it, so KVCompass transfers KV cache replicas within regions and recomputes shipped prefixes across regions. Prompts contain only a few bytes per token, making their transfer cost small. Registration uses an insert; migration targets idle workers to limit interference with serving requests.

Queue-view propagation traffic. A region's queue vector holds one b -bit estimate per worker, giving an average vector size of $(W/R)b$ bits. Under broadcast alone, each region sends its vector to the other $R - 1$ regions every Δ , averaging $(R - 1)(W/R)b$ bits per region per period. Across the pool, this costs $(R - 1)Wb/\Delta$ bits/s, including unequal regional worker counts. Piggybacking carries vectors on existing WAN responses, and suppressing broadcasts to regions that received a piggybacked vector in the current period can reduce traffic below this bound.

4 Implementation

As shown in Figure 10, we implement KVCompass in two forms: a real-machine prototype for end-to-end validation and an event-driven simulator for large-scale evaluation. The real-machine prototype couples instrumented SGLang workers with the KV cache discovery plane, while the simulator reuses the discovery-plane logic and models workers and network execution as events.

Real-machine prototype. The prototype consists of SGLang workers and the KV cache discovery plane. SGLang workers are built on SGLang v0.5.10. Each worker serves one model replica deployed on one or more GPUs, provides request scheduling and model execution, manages the two-tier KV cache through admission, demotion, promotion, and eviction, and exposes a control module for load reporting, KV cache state-event reporting, and replica migration coordination. Cross-region requests are forwarded over the real WAN to the selected worker.

The discovery plane builds on the reference C++ Ludo implementation [40] and comprises multi-level lookup units at edge nodes, sharded maintenance units, and the joint-score forwarding logic. Maintenance units process worker-reported KV cache events and propagate the resulting state updates to the lookup units. Edge nodes then rank workers with the joint score (Section 3.4) under their local view.

Simulator. The simulator reuses the same lookup, maintenance, and cache-state logic, while replacing physical workers and the real network with their event-driven counterparts. Its request loop advances a virtual clock, replays requests in arrival order, and, for each request, models forwarding, queueing, service, and subsequent KV cache state updates. The simulated worker pool captures queueing delay, prefill and decode time, inter-region network latency, and the two-tier VRAM and DRAM KV cache, with prefill, decode, and WAN costs calibrated to hardware benchmarks and network measurements. This calibration and the shared discovery-plane logic connect the simulator to the prototype, but we have not quantified simulation error against a testbed run under identical settings.

5 Evaluation

In this section, we evaluate TTFT latency, end-to-end latency and end-to-end service throughput both on the real-machine testbed (Section 5.2) and large-scale simulation (Section 5.3). We also evaluate load balancing, reported as each worker's load relative to the mean load across all workers.

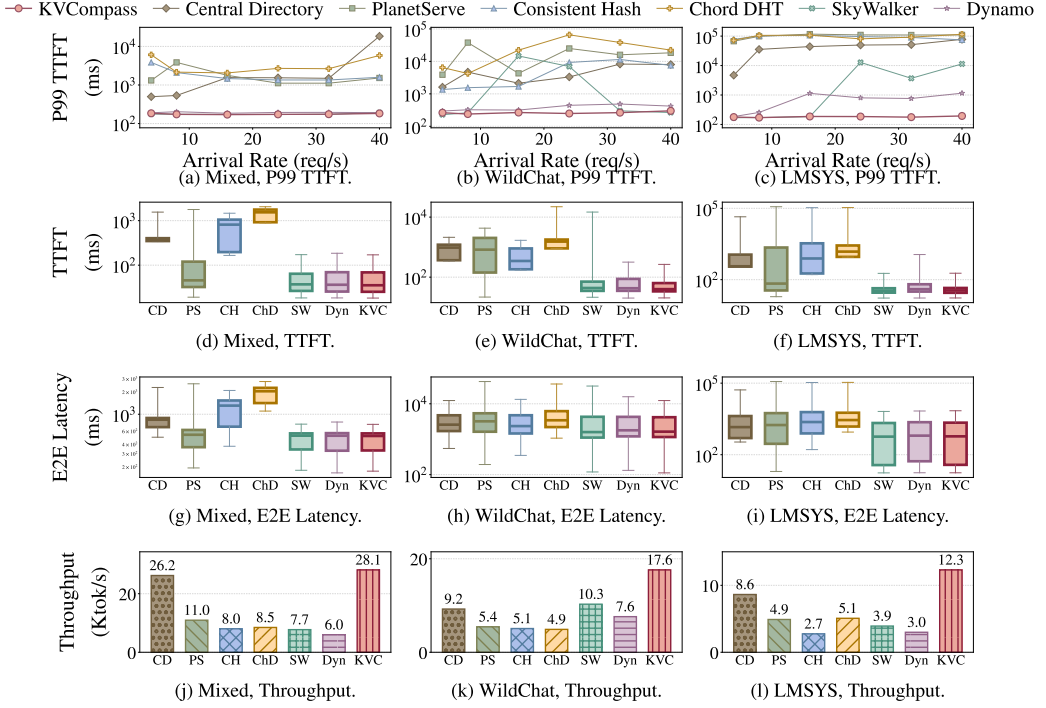


Fig. 11. P99 TTFT vs. arrival rate, TTFT and end-to-end latency distributions, and throughput on Mixed, WildChat, and LMSYS, measured on the 14 workers across three-continent real-machine testbed (8/2/4 workers in US/EU/Asia) over the public Internet.

We then quantify the contribution of each mechanism by ablation study (Section 5.4), measure the cost of every index operation (Section 5.5), and measure index memory usage (Section 5.6). Appendix A reports larger-scale results with 128 workers across 16 regions at arrival rates up to 18000 req/s. Appendix B reports additional testbed results across four North-American regions.

5.1 Experimental Methodology

Workload. We evaluate on three different workloads to reflect different deployment scenarios. *Mixed* consists of three datasets: ToolBench [14] for agent tool-calling traces (50%), LooGLE [23] for long-document QA (5%) and APPS [16] for competitive-programming problems (45%). *WildChat* [55] consists of real user conversations collected from ChatGPT. *LMSYS* [57] consists of real-world conversations collected from the Chatbot Arena platform.

Eviction policy. The discovery plane does not control each worker’s local KV cache eviction policy. The results of Figure 6 show that the best eviction policy differs across different workloads. To isolate discovery and forwarding gains from eviction effects, we use LFU for Mixed and LRU for WildChat and LMSYS, the best policy for each workload.

Baselines. We compare KVCompass with six baselines: (1) Consistent Hashing, using six independent content-hash rings; (2) Chord [43], using the six-level hash-based layout with one DHT key per chunk-size level; (3) PlanetServe [9] using hash radix tree; (4) Central Directory, a centralized coordinator with global KV cache state; (5) SkyWalker [48], which deploys a load balancer on each region and forwards a request to a replica only when the replica’s pending queue is empty using

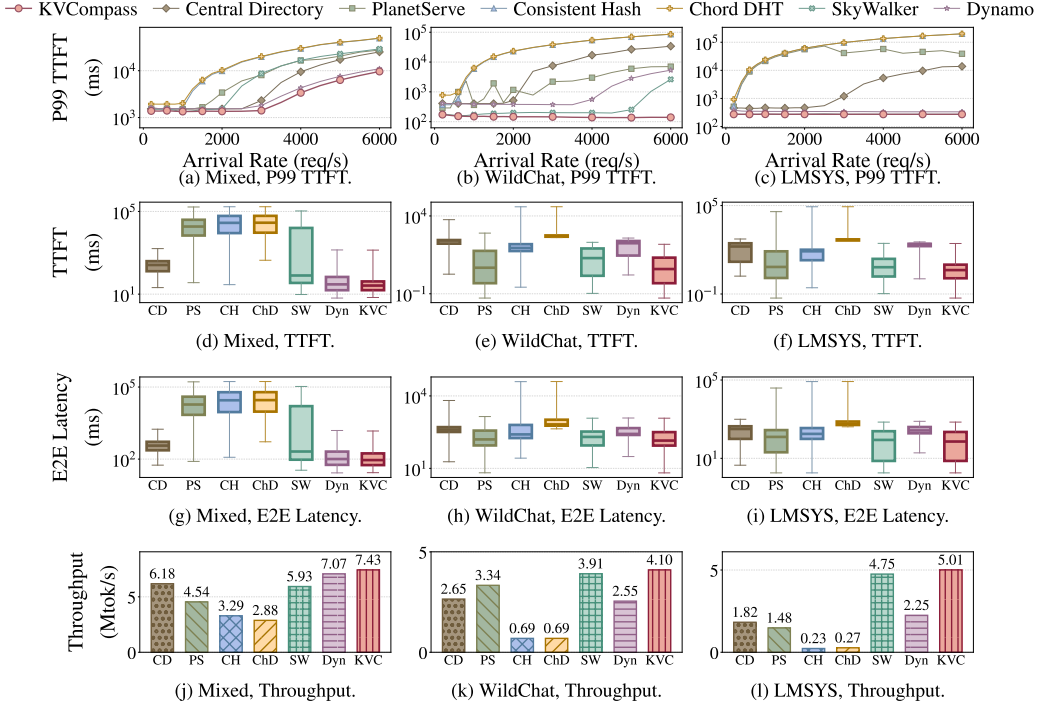


Fig. 12. P99 TTFT vs. arrival rate, TTFT and end-to-end latency distributions, and throughput on Mixed, WildChat, and LMSYS, simulated on 64 workers across 8 regions.

“selective pushing” mechanism, and (6) Dynamo [32], which deploys a KV cache-aware dynamo router implemented by Nvidia on each region and periodically broadcasts to get the global view. **Configuration.** The testbed and the simulator share the same configurations. The queue-estimate broadcast period is $\Delta = 100$ ms with EWMA weight $w = 0.5$, and the capacity price is capped at $\lambda_{\max} = 8$. Migration triggers at $H = 3$ misses of a prefix family within a $T_h = 1$ s sliding window.

5.2 Testbed Validation

Testbed Setup. We use up to 14 workers on three Vast.ai¹ multi-GPU servers across three regions, with 8 RTX 5090 GPUs in the US, 4 in Asia, and 2 in Europe. Each GPU hosts one SGLang [58] worker serving Llama-3.1-8B-Instruct (BF16). Within each region, the edge dispatches requests over kernel loopback TCP. Across regions, all traffic traverses the public Internet, with measured one-way latencies of 160 ms between US and Europe, 126 ms between US and Asia, and 149 ms between Europe and Asia. KVCompass derives indexed lengths at startup from the calibrated RTX 5090 prefill speed of 0.085 ms/token.

P99 TTFT under increasing load. Figure 11 (a)(b)(c) show P99 TTFT at arrival rates of 4–40 req/s. At 16 req/s, KVCompass is **8.6–12×** lower than the hash-placement schemes and the Central Directory on Mixed, **6.3–83×** lower on WildChat, and **236–626×** lower on LMSYS. Hash-placement schemes route by prefix hash, concentrating popular-prefix requests on a few workers and overloading them. The Central Directory pays a WAN round trip to a single coordinator per request, increasing TTFT. The other two baselines match KVCompass at low load but degrade as

¹<https://vast.ai>

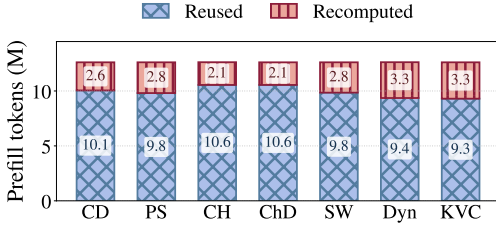


Fig. 13. Reused versus recomputed prefill tokens by scheme on the real-machine testbed (Mixed).

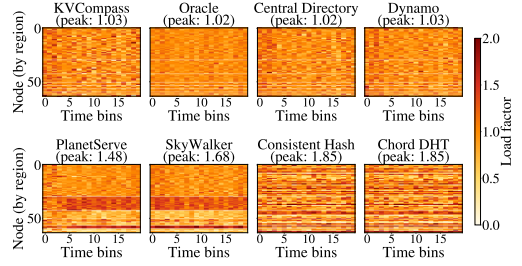


Fig. 14. Per-worker load factor heatmap across schemes. Peak: the hottest worker's time-averaged load factor, relative to each scheme's balanced mean.

the rate increases. SkyWalker's P99 TTFT spikes to $55\times$ above KVCompass on WildChat at 16 req/s and $69\times$ on LMSYS at 24 req/s. SkyWalker considers only queue occupancy, sending requests away from KV cache holders to pay full prefill time elsewhere. Dynamo is $4\text{--}6\times$ above KVCompass on LMSYS from 16 req/s onward because its broadcast global view becomes stale between refreshes.

Latency distributions. Figure 11 (d)(e)(f) show TTFT latency box plots at 16 req/s arrival rate. On WildChat, SkyWalker's P99 TTFT is $55\times$ higher. Dynamo stays much closer, $1.2\times$ higher. For end-to-end latency at 16 req/s rate, Figure 11(g)(h)(i) show that KVCompass's P99 is $3.5\times$ lower than PlanetServe, $3.7\times$ lower than Chord, and $3.1\times$ lower than the Central Directory on Mixed.

Service throughput. We show the service throughput in Figure 11 (j)(k)(l). KVCompass keeps the highest throughput on all three workloads. On Mixed, the Central Directory comes within 7% of KVCompass. The other baselines deliver $2.6\text{--}4.7\times$ lower throughput. On WildChat, KVCompass delivers $1.7\times$ the throughput of SkyWalker, and $1.9\text{--}3.6\times$ that of the rest. On LMSYS, KVCompass delivers $1.4\times$ the throughput of the Central Directory and $2.4\text{--}4.5\times$ that of the rest. Every recomputed prefix consumes GPU time that could have served new requests, and concentrating load onto a few workers wastes the computation of the idle rest. KVCompass jointly considers KV cache reuse and load across all workers, achieving the highest throughput.

Reused versus recomputed tokens. Figure 13 shows each scheme's prefill tokens split into those reused from a KV cache hit and those recomputed from scratch. KVCompass does not reuse the most. The hash-placement schemes reuse the largest share, 10.6 M tokens against KVCompass's 9.3 M on Mixed, because they always route a prefix to the same worker, and KVCompass recomputes somewhat more. KVCompass nonetheless delivers the highest throughput of all schemes (Figure 11), so its gains come mainly from load-aware forwarding rather than maximizing cache reuse.

5.3 Large-Scale Simulation

We scale the same Mixed workload to 50 independent tenants each replaying the workload with a unique private system prompt, producing ≈ 375 K distinct prefix families across 64 workers in 8 geographic regions. Per-region capacity follows a real LLM-deployment census [17], so the worker distribution matches a measured deployment, giving 30 workers in the United States, 13 in China, 8 in Germany, and 2–3 in each of Japan, Singapore, the United Kingdom, South Korea, and Australia. The 30-worker United States region uses 64-bit discovery index values, while the other regions use 32-bit values. Each worker is modeled as an A100-80GB GPU running Llama-3.1-8B-Instruct [12](BF16), with 32 GB VRAM and 128 GB host DRAM reloadable over PCIe 4.0 at ≈ 25 GB/s. Inter-region WAN latency follows a realistic matrix (12–150 ms RTT) derived from CloudPing.co AWS measurements, the wide-area latencies a real inter-region deployment would

see. Prefill and decode costs are calibrated against published A100 BF16 benchmarks on vLLM [21] and SGLang [58], so per-token compute times match real hardware. Requests are replayed at 200–6000 req/s. We compare to the same six baselines as the real-machine testbed.

Latency and throughput. Figure 12 plots P99 TTFT versus arrival rate and latency distribution at 2750 req/s. At 3000 req/s on Mixed, KVCompass achieves P99 TTFT **1.6×** lower than the Central Directory, **5.7×** lower than PlanetServe, **6.2×** lower than SkyWalker, **14×** lower than the DHT schemes, and **1.3×** lower than Dynamo. Under sustained overload at 6000 req/s the advantage holds at **1.1–5.1×** across all baselines. The box panels show that the advantage scales to 64 workers: across the three workloads at 2750 req/s, the DHT schemes land **125–312×** above KVCompass on P99 TTFT and **33–118×** on P99 end-to-end latency, PlanetServe lands **5.2–145×** and **1.2–111×** above, and the Central Directory lands **1.2–38×** and **1.2–5.4×** above. Among the load-aware baselines, SkyWalker sits **78×** above KVCompass on P99 TTFT on Mixed and Dynamo **2.5×** above on WildChat. The throughput panels show that KVCompass delivers the highest throughput on all three workloads: about **5%** above Dynamo and SkyWalker, and **2.3–22×** above the hash-placement schemes. KVCompass keeps both its latency and its throughput advantages in large-scale simulation.

Load Balancing. A worker’s load factor measures how far it sits from an even split of the load. Over a time window, we sum the GPU compute time of the requests sent to each worker and divide it by the average over all workers. Figure 14 maps each worker’s load factor across the simulated 64 workers over a 30-second window at the 2750 req/s arrival rate on the Mixed workload. Each scheme’s peak is its hottest node’s time-averaged load factor relative to its balanced mean.

The Oracle forwards every request to the least-loaded worker in the whole worker pool, so its peak of **1.02** is the best lower bound any scheme can achieve. KVCompass peaks at **1.03**, within **2%** of the Oracle. Our discovery plane considers the current estimated queueing time to every worker globally. The Central Directory and Dynamo balance comparably, peaking at 1.02 and 1.03, since the Central Directory holds exact global state and the Dynamo routers on each region are load-aware. SkyWalker peaks at 1.68, since its locality-first policy keeps every request inside its local region as long as any local replica is free. The hash-placement schemes peak at 1.85 and PlanetServe at 1.48, concentrating the requests for popular prefixes on a few hash-designated workers, the same failure seen in Section 5.2. Read together with the reuse breakdown of Figure 13, this suggests that in these experiments a better load balance that avoids these hot spots contributes more to performance than maximizing reused KV cache does.

5.4 Ablation Study

In order to evaluate each mechanism of KVCompass and quantify their contribution, we disable one mechanism at a time and measure closed-loop service throughput under the same real-machine testbed setting, on the Mixed workload.

Figure 15 shows the results. No-loadaware removes estimated queueing time from the joint score, leaving only KV cache affinity and network latency. Throughput drops by 34% as requests concentrate on workers holding hot-prefix KV caches. No-xregion-reuse disables cross-region lookups and forwarding. Throughput drops by 10% because requests recompute prefixes locally even when remote KV cache reuse would save more time than the added network latency. Tier-blind removes KV cache tier information from the discovery index, pricing DRAM hits like VRAM hits.

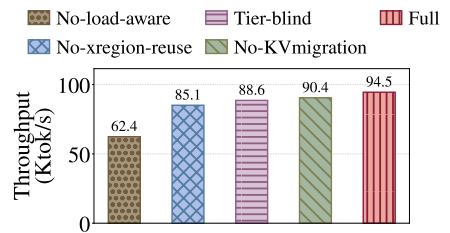


Fig. 15. Ablation study. Service throughput with one mechanism removed at a time on the real-machine testbed (Mixed workload).

Table 2. Cost of each discovery index operation. Write-path ranges span 85–95% table occupancy.

Operation	P50 Time	P99 Time	Rate
Lookup	310 ns	590 ns	5.4 M/s
Forwarding	20.6 μ s	21.5 μ s	48 K/s
Insert	1.2–1.4 μ s	4.4–6.1 μ s	0.6–0.7 M/s
Update	170 ns	1.0–1.8 μ s	4.8–5.8 M/s
Remove	0.8–0.9 μ s	4.8–5.9 μ s	0.8–0.9 M/s

Table 3. Discovery index memory at 10^9 entries.

Data Structure	Bits/entry	Total
KVCompass	48	6.0 GB
Hash map directory	498	62 GB
Token radix tree	12,900	1.6 TB
PlanetServe radix tree	73	9.2 GB

Throughput drops by 6% because ignoring DRAM reload time can favor slower workers. No-KVMigration disables proactive migration of hot KV caches. Throughput drops by 4% as requests either recompute missing prefixes or pay extra network latency to reach a holder.

5.5 Operations Microbenchmarks

We benchmark discovery index operations on a dual 96-core CPU host of the real-machine testbed. Table 2 reports P50 and P99 latency and single-thread throughput, with write operations measured at 85–95% table occupancy.

Lookup and Forwarding. A lookup in a view of 10^7 entries takes 310 ns at the P50 and 590 ns at the P99. For a 4096-token prompt, forwarding queries a five-level Ludo table and scores all 14 workers, taking 20.6 μ s at the P50 and 21.5 μ s at the P99, with 48 K decisions per second on one thread. These costs add little overhead to request serving.

Insert. P50 insert latency rises from 1.2 μ s at 85% occupancy to 1.4 μ s at 95%, while P99 rises from 4.4 to 6.1 μ s. Fuller tables increase the chance of hitting occupied slots and displacing entries.

Update. An update rewrites one 2-bit field in the prototype’s 32-bit value. It is the cheapest write operation, taking 170 ns at the P50 across the tested occupancies. P99 latency is 1.0–1.8 μ s, with throughput of 4.8–5.8 M updates per second.

Remove. P50 remove latency is 0.8–0.9 μ s, close to insertion because clearing a slot follows the same path as writing it. P99 latency is 4.8–5.9 μ s, with throughput of 0.8–0.9 M removals per second. Table 2 gives the remaining latency and throughput results.

5.6 Discovery Index Memory Footprint

Table 3 compares memory footprints at 10^9 entries. The benchmark uses 14 workers and 32-bit values for KVCompass, with real Mixed token sequences at smaller scales. At 10^9 entries, we directly allocate and write to the lookup view’s bucket and classifier arrays, measuring **48 bits per entry** and **6.0 GB** without inserting all prefixes. The difference from the theoretical 37 bits per entry includes allocation overhead. The hash map and token radix tree estimates are extrapolated from smaller runs, giving 498 bits per entry and 1.6 TB, respectively. PlanetServe’s hash radix tree uses 9.2 GB in a full-scale run. These KVCompass results apply to the measured 32-bit layout. Tables with 64-bit values use more memory per entry, and total memory depends on the entry count and allocation overhead of all tables.

5.7 Robustness to WAN and Control-Plane Variation

We evaluate WAN latency jitter, discovery index update loss, and update delay in the large-scale simulation on Mixed at 3000 req/s. Actual WAN latency varies around the static latency matrix used by the joint score. Table 4 shows that $\pm 100\%$ jitter preserves the KV cache hit rate and changes P99 TTFT by less than 2%. With 30% update loss, the hit rate drops by 2.5 percentage points and P99 TTFT rises by 4.2%. A 300 ms update delay changes the hit rate by 0.3 percentage points and P99

Table 4. KVCompass sensitivity to WAN jitter and update loss/delay (Mixed, 3000 req/s).

	Base	Jitter ($\pm\%$)			Loss (%)			Delay (ms)		
	0	25	50	100	10	20	30	100	200	300
P99 TTFT (ms)	1370	1355	1345	1345	1328	1404	1427	1343	1324	1324
Hit rate (%)	43.8	43.8	43.8	43.8	42.7	42.3	41.3	42.9	43.4	44.1

Table 5. False hits in the keyless discovery index for never-inserted keys and real requests.

	Never-inserted key, by table load					End-to-end, by workload		
	20%	40%	60%	80%	95%	Mixed	WildChat	LMSYS
False-positive rate (%)	9.7	19.1	28.6	38.2	45.3	0.10	1.27	0.34

TTFT by 3.4%. Inaccurate latency estimates and stale entries may reduce reuse or increase latency, but the selected worker uses only the KV cache it holds and recomputes the rest. The prototype does not automatically retransmit failed update batches. Later worker reports or reloading the maintenance units' lookup view may correct stale entries, with no guaranteed recovery time.

5.8 False Hits in the Keyless Discovery Index

The edge lookup view stores no keys, so a query for a never-inserted prefix can return a false locator from an occupied Ludo slot. Table 5 reports two rates. The structural false-positive rate measures how often a never-inserted key returns a false locator, reaching 45.3% at 95% table load. The end-to-end false-lookup rate measures how often the worker selected for a real request lacks the claimed KV cache match, giving 0.10% on Mixed, 1.27% on WildChat, and 0.34% on LMSYS. These rates use different query populations and measure different stages. Real requests can query prefixes already in the discovery index, and a false locator affects forwarding only if its worker wins the joint score. Cache belief discounts the claimed matches of workers that repeatedly reuse fewer tokens than claimed. A false lookup may reduce KV cache reuse, but the selected worker recomputes any missing prefix, preserving answer correctness.

6 Discussion

Serving multiple models. Our main evaluation serves one model across all workers, while Internet-scale deployments may serve several models. Even for the same prefix, different models, precisions, or parallel partitions produce incompatible KV tensors [11, 28]. Including model identity and configuration in the discovery index key would keep these KV caches separate. A lookup would then return only workers holding KV caches compatible with the request's model and configuration. Entries for different models could coexist in one table without changing the discovery index structure, operations, or lookup views. The orchestration layer would still decide which workers load each model and when to replace its weights [10, 52].

Prefill-decode disaggregation. Our main evaluation runs prefill and decode on the same worker. Disaggregated systems separate the two phases because prefill is compute-bound and decode is bound by memory bandwidth [34, 36, 59]. In this setting, KVCompass routes prefill requests and tracks KV caches on prefill workers, since reuse saves prefill computation. Decode output becomes part of the next turn's prefix and enters the discovery index when that prefix is prefilled. The serving stack selects decode workers and transfers KV caches. Appendix C evaluates this integration on a three-region testbed with SGLang and Mooncake, where KVCompass achieves lower median and

P99 TTFT than all evaluated baselines. Decode scheduling, transfer scheduling, and pool sizing remain the responsibility of the disaggregated serving stack.

Security and privacy. KV cache reuse requires matching the entire indexed prefix, so a lookup does not return another user's unknown prompt content. However, timing differences can reveal whether a known or guessed prefix is cached [9, 13, 41]. Existing timing attacks extend guesses a few tokens at a time. In KVCompass, indexed prefixes span at least hundreds of tokens, which may make incremental guessing more difficult, although an attacker who knows a suspected prefix may still infer its presence. Including a user or user-group identity in the discovery index key would restrict KV cache sharing to that group. The provider could choose the sharing boundary, while timing side channels within each group may still need mitigations such as response padding [2].

7 Related Work

Geo-distributed LLM serving. KDN [6] and Prefill-as-a-Service [35] motivate KV cache sharing across datacenters. PlanetServe [9] supports decentralized serving with a radix-tree index for regional prefix reuse, while SkyWalker [48] balances load across regions. GORGO [44] optimizes cross-region routing using KV cache locality, network latency, and queue state, an objective close to KVCompass's joint score. Its KV cache index estimates prefix availability without directly observing worker evictions. KVCompass instead tracks KV cache holders and storage tiers through KV cache events and replicates compact lookup views at every edge node. Our main contribution is how reusable KV caches are discovered across regions.

Intra-cluster prefix caching. SGLang [58], ChunkAttention [50], and Hydragen [19] reuse shared prefix KV cache, while CacheBlend [49] reuses KV cache for arbitrary chunks. Mooncake [36], MemServe [18], and Infinite-LLM [24] pool KV cache across storage tiers. DualMap [54], Helix [31], Preble [42], LMCache [26], and LoongServe [47] support KV cache reuse and scheduling within a cluster. KVCompass's joint score builds on prior KV cache-aware and load-aware routing, while its discovery plane finds KV caches across regions at prefix lengths chosen from network latency.

Distributed naming and discovery. Chord [43], Kademlia [30], Pastry [39], and IPFS [5] map an item's identity to a hash-designated node that holds or routes to it. KV caches are created on the workers that serve requests, so their locations cannot be derived from prefix hashes. KVCompass decouples KV cache identity from placement by tracking holders wherever KV caches are created.

Compact key-value lookup structures. Othello [53] provides keyless $O(1)$ lookups, Ludo [40] adds dynamic updates, and Smash [29] applies explicit-write lookup to distributed storage placement. Cuckoo filters [8] address set membership. KVCompass tracks changing KV cache holders and storage tiers at latency-derived prefix lengths for Internet-scale LLM serving.

8 Conclusion

This paper presents KVCompass, a KV cache discovery plane for Internet-scale prefix sharing in LLM serving. By decoupling KV cache identity from placement, KVCompass discovers reusable KV caches wherever workers create them. Hash-placement schemes cannot provide this decoupling because they determine KV cache placement from its identity. Compact local lookups and a joint score balance prefix reuse, worker load, and network distance. Real-machine experiments and large-scale simulations show that KVCompass achieves lower TTFT and end-to-end latency and higher throughput than all baselines, while maintaining good load balance.

Acknowledgments

The authors were partially supported by NSF Grants 2322919, 2420632, 2426031, and 2426940. We thank our shepherd and the anonymous reviewers for their helpful feedback.

References

- [1] Anthropic. 2024. Prompt Caching. <https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching>.
- [2] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) (CCS '10). Association for Computing Machinery, New York, NY, USA, 297–307. doi:10.1145/1866307.1866341
- [3] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23, 2 (March 2003), 22–28. doi:10.1109/MM.2003.1196112
- [4] Djamel Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2009. Hash, Displace, and Compress. In *Algorithms - ESA 2009, 17th Annual European Symposium (Lecture Notes in Computer Science, Vol. 5757)*. Springer, 682–693. doi:10.1007/978-3-642-04128-0_61
- [5] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *ArXiv abs/1407.3561* (2014). <https://api.semanticscholar.org/CorpusID:15655499>
- [6] Yihua Cheng, Kuntai Du, Jiayi Yao, and Junchen Jiang. 2024. Do Large Language Models Need a Content Delivery Network? *arXiv:2409.13761 [cs.DC]* <https://arxiv.org/abs/2409.13761>
- [7] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. doi:10.1145/2408776.2408794
- [8] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.
- [9] Fei Fang, Yifan Hua, Shengze Wang, Ruilin Zhou, Yi Liu, Chen Qian, and Xiaoxue Zhang. 2026. {PlanetServe}: A Decentralized, Scalable, and {Privacy-Preserving} Overlay for Democratizing Large Language Model Serving. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2111–2129.
- [10] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.
- [11] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. *Proceedings of Machine Learning and Systems* (2024).
- [12] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).
- [13] Chenchen Gu, Xiang Lisa Li, Rohith Kuditipudi, Percy Liang, and Tatsunori Hashimoto. 2025. Auditing prompt caching in language model APIs. In *Proceedings of the 42nd International Conference on Machine Learning* (Vancouver, Canada) (ICML '25). JMLR.org, Article 794, 20 pages.
- [14] Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics, Bangkok, Thailand, 11143–11156. <https://aclanthology.org/2024.findings-acl.664>
- [15] Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.
- [16] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).
- [17] Xinyi Hou, Jiahao Han, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2026. Mapping the Landscape of LLM Deployment in the Wild: Prevalence, Patterns, and Perils. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 1, Article 15 (March 2026), 26 pages. doi:10.1145/3788097
- [18] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. *arXiv preprint arXiv:2406.17565* (2024).
- [19] Jordan Juravsky, Bradley Brown, Ryan Saul Ehrlich, Daniel Y Fu, Christopher Re, and Azalia Mirhoseini. 2024. Hydragen: High-Throughput LLM Inference with Shared Prefixes. In *Workshop on Efficient Systems for Foundation Models II @ ICML2024*. <https://openreview.net/forum?id=Ye3ia34Fpp>
- [20] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.
- [21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*. 611–626.

- [22] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 9459–9474. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf
- [23] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2024. LooGLE: Can Long-Context Language Models Understand Long Contexts?. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 16304–16333. doi:10.18653/v1/2024.acl-long.859
- [24] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, Zhigang Ji, Yong Li, and Wei Lin. 2024. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. *arXiv preprint arXiv:2401.02669* (2024).
- [25] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: efficient serving of LLM-based applications with semantic variable. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (Santa Clara, CA, USA) (OSDI'24)*. USENIX Association, USA, Article 50, 17 pages.
- [26] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. 2025. Lmcache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665* (2025).
- [27] 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. 2026. DroidSpeak: KV Cache Sharing Across Fine-tuned Model Variants. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 319–338. <https://www.usenix.org/conference/nsdi26/presentation/liu-yuhan>
- [28] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. 2024. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 38–56.
- [29] Yi Liu, Shouqian Shi, Minghao Xie, Heiner Litz, and Chen Qian. 2025. Smash: Flexible, Fast, and Resource-Efficient Placement and Lookup of Distributed Storage. *Commun. ACM* 68, 10 (2025), 113–121.
- [30] Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International workshop on peer-to-peer systems*. Springer, 53–65.
- [31] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2024. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow. *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (2024). <https://api.semanticscholar.org/CorpusID:270217361>
- [32] NVIDIA Corporation. 2025. NVIDIA Dynamo: A Datacenter Scale Distributed Inference Serving Framework. <https://github.com/ai-dynamo/dynamo>. Accessed: 2026-07-07.
- [33] OpenAI. 2024. Prompt Caching. <https://platform.openai.com/docs/guides/prompt-caching>.
- [34] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132. doi:10.1109/ISCA59077.2024.00019
- [35] Ruoyu Qin, Weiran He, Yaoyu Wang, Zheming Li, Xinran Xu, Yongwei Wu, Weimin Zheng, and Mingxing Zhang. 2026. Prefill-as-a-Service: KVCache of Next-Generation Models Could Go Cross-Datacenter. *arXiv preprint arXiv:2604.15039* (2026).
- [36] Ruoyu Qin, Zheming Li, Weiran He, Jiale Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 155–170. <https://www.usenix.org/conference/fast25/presentation/qin>
- [37] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (San Diego, California, USA) (SIGCOMM '01)*. Association for Computing Machinery, New York, NY, USA, 161–172. doi:10.1145/383059.383072
- [38] Reuters. 2023. Tech Giants' AI-like Bing, Bard Poses Billion-Dollar Search Problem. <https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/>. Accessed: 2023-09-21.
- [39] Antony I. T. Rowstron and Peter Druschel. 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (Middleware '01)*. Springer-Verlag, Berlin, Heidelberg, 329–350.

- [40] Shouqian Shi and Chen Qian. 2020. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 2 (2020), 1–32.
- [41] Linke Song, Zixuan Pang, Wenhao Wang, Zihao Wang, XiaoFeng Wang, Hongbo Chen, Wei Song, Yier Jin, Dan Meng, and Rui Hou. 2025. The Early Bird Catches the Leak: Unveiling Timing Side Channels in LLM Serving Systems. *IEEE Transactions on Information Forensics and Security* 20 (2025), 11431–11446. doi:10.1109/TIFS.2025.3622954
- [42] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiyang Zhang. 2025. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=meKEKDhdx>
- [43] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review* 31, 4 (2001), 149–160.
- [44] Alessio Ricci Toniolo, Rome Thorstenson, and Abinaya Dinesh. 2026. GORG: Online Tuning for Cross-Region Network-Aware LLM Serving. arXiv:2602.11688 [cs.NI] <https://arxiv.org/abs/2602.11688>
- [45] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. {KVCache} Cache in the Wild: Characterizing and Optimizing {KVCache} Cache at a Large Cloud Provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 465–482.
- [46] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 31–31. doi:10.1109/SC.2006.19
- [47] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (Austin, TX, USA) (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 640–654. doi:10.1145/3694715.3695948
- [48] Tian Xia, Ziming Mao, Jamison Kerney, Ethan J. Jackson, Zhifei Li, Jiarong Xing, Scott Shenker, and Ion Stoica. 2026. SkyWalker: A Locality-Aware Cross-Region Load Balancer for LLM Inference (EUROSYS '26). Association for Computing Machinery, New York, NY, USA, 1349–1364. doi:10.1145/3767295.3769353
- [49] Jiayi Yao, Hanchen Li, Yuhua Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*.
- [50] Lu Ye, Zewei Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. In *Annual Meeting of the Association for Computational Linguistics*. <https://api.semanticscholar.org/CorpusID:267897412>
- [51] Gyeong-In Yu and Joo Seong Jeong. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *USENIX Symposium on Operating Systems Design and Implementation*. <https://api.semanticscholar.org/CorpusID:251734964>
- [52] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, Ion Stoica, Harry Xu, and Ying Sheng. 2025. Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving. *arXiv preprint arXiv:2505.04021* (2025).
- [53] Ye Yu, Djamel Belazzougui, Chen Qian, and Qin Zhang. 2018. Memory-Efficient and Ultra-Fast Network Lookup and Forwarding Using Othello Hashing. *IEEE/ACM Transactions on Networking* 26, 3, 1151–1164. doi:10.1109/TNET.2018.2820067
- [54] Ying Yuan, Pengfei Zuo, Bo Wang, Zhangyu Chen, Zhipeng Tan, and Zhou Yu. 2026. DualMap: Enabling Both Cache Affinity and Load Balancing for Distributed LLM Serving. In *The Fourteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=zCadrJ32Xn>
- [55] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. WildChat: 1M ChatGPT Interaction Logs in the Wild. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=Bl8u7ZRlbM>
- [56] Anastasiya Zharovskikh. 2023. Applications of Large Language Models. <https://indatalabs.com/blog/large-language-model-apps>. Accessed: 2023-09-21.
- [57] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=BofDKxfwt0>
- [58] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

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

A Simulation at Larger Scale

We extend the large-scale simulation of Section 5.3 to 128 workers across 16 geographic regions, using the same workloads and baselines with 50 independent tenants per workload. Per-region capacity follows the same real LLM-deployment census [17], giving 26 and 14 workers in two United States regions, 12 in China, 10 each in Germany and Ireland, 8 each in Japan and the United Kingdom, and 2 to 7 in each of Singapore, South Korea, India, Australia, Brazil, Canada, Italy, Bahrain, and South Africa. Each worker is modeled as an A100-80GB GPU running Llama-3.1-8B-Instruct [12] (BF16), with 32 GB VRAM and 128 GB host DRAM reloadable over PCIe 4.0 at ≈ 25 GB/s. Across region WAN latency follows the same CloudPing.co-derived matrix extended to the added regions by great-circle distance, with one-way latencies spanning 9 to 144 ms across the 16 regions. Requests are replayed at 400–18000 req/s, scaled up with the doubled capacity.

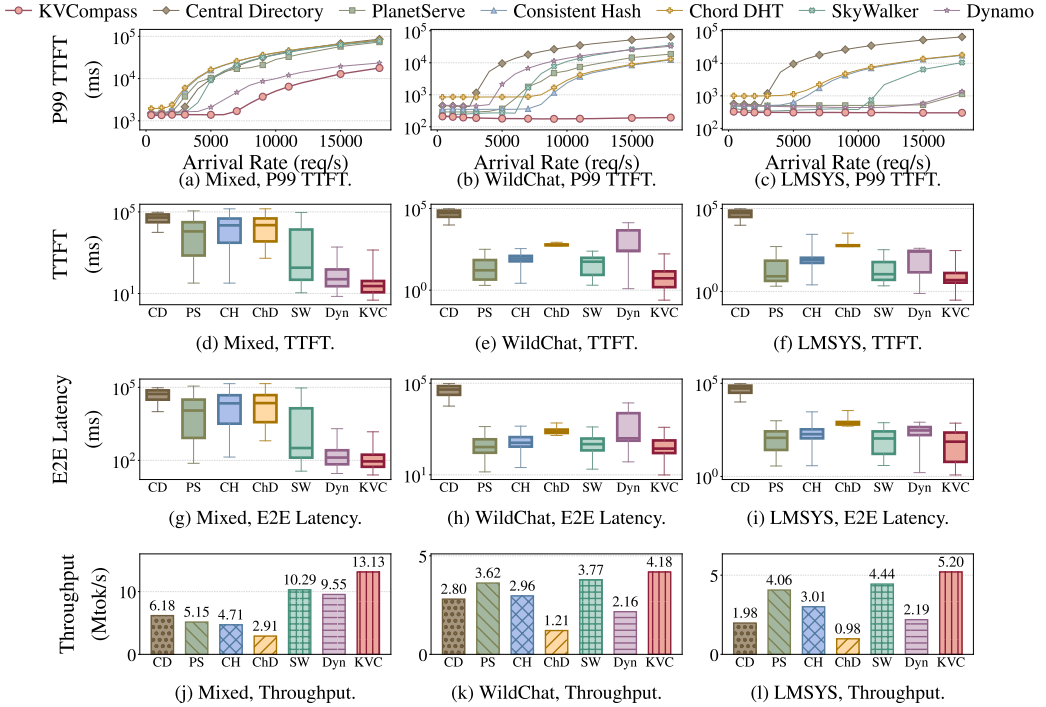


Fig. 16. P99 TTFT vs. arrival rate, TTFT and end-to-end latency distributions, and throughput on Mixed, WildChat, and LMSYS, simulated on 128 workers across 16 regions.

Latency and throughput. Figure 16 plots P99 TTFT versus arrival rate and latency distribution at 5500 req/s. Latency distributions use 500,000 requests per workload. At 12000 req/s on Mixed, KVCompass achieves P99 TTFT $1.8\text{--}6.5\times$ lower than every baseline; the gap widens to $27\text{--}218\times$ on WildChat and $1.4\text{--}128\times$ on LMSYS. KVCompass also delivers the highest throughput on all three workloads, 13.1, 4.2, and 5.2 Mtok/s, $1.1\text{--}1.3\times$ above the closest load-aware baseline and $1.4\text{--}5.3\times$ above the hash-placement schemes. At twice the region count, worker count, and arrival rate of

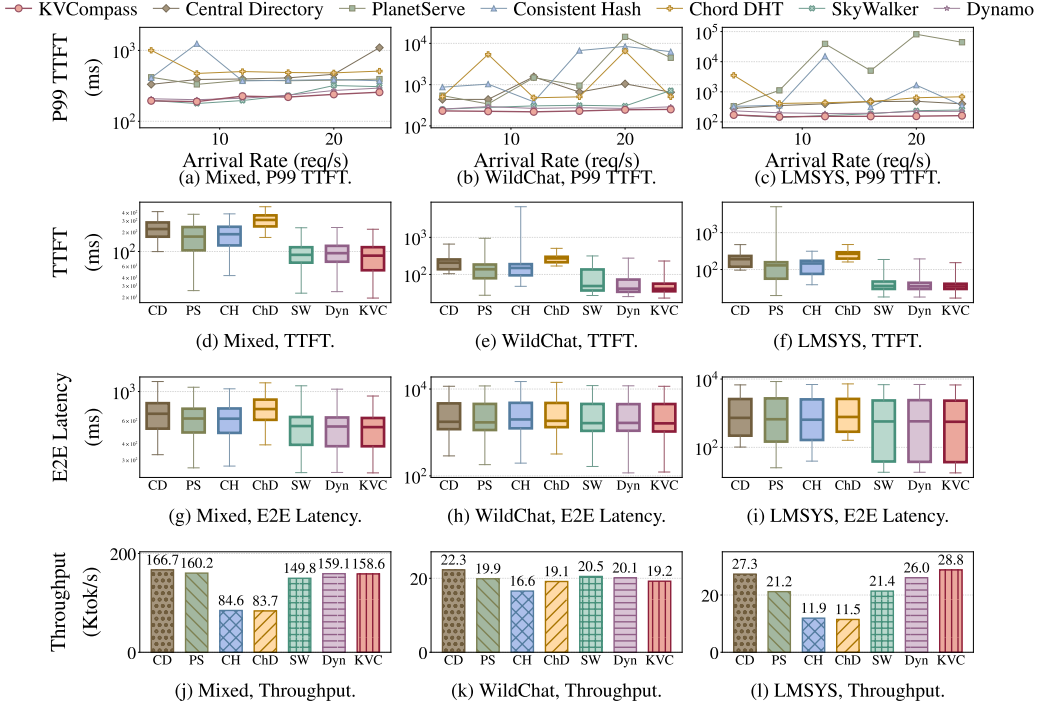


Fig. 17. P99 TTFT vs. arrival rate, TTFT and end-to-end latency distributions, and throughput on Mixed, WildChat, and LMSYS, on an 11-worker, four-region North-American testbed over the public Internet.

the main simulation, KVCompass still holds the best P99 TTFT, tail latency, and throughput on every workload, preserving its advantage at scale.

B Results over a Smaller Region Range

The testbed of Section 5.2 spans three continents. To check that the discovery plane holds up under a very different geography, we also ran the full evaluation on a second testbed over a much smaller region range. It uses 11 RTX 5090 workers across four North-American regions, with 2 GPUs in Pennsylvania, 4 in Ohio, 1 in New York, and 4 in Calgary, and one-way WAN latencies of 5.8 to 16.6 ms among the three eastern regions and 24.9 to 37.4 ms to Calgary, far shorter than the intercontinental links of Section 5.2. Figure 17 reports the same three workloads and seven schemes. At 16 req/s, KVCompass achieves P99 TTFT 1.7 to 2.2× below the hash-placement schemes and the Central Directory on Mixed and up to 29 to 33× on WildChat and LMSYS, and maintains competitive throughput. The advantage carries over to this much closer region range, which shows the design is robust and not tuned to one geography.

C Prefill-Decode Disaggregation

KVCompass assumes prefill and decode run on the same worker, but serving systems increasingly disaggregate the two phases onto separate pools [34, 36, 59]. KVCompass adapts naturally, because KV cache reuse saves prefill computation, so the discovery plane routes only the prefill request and indexes only the prefill workers, while the decode instance and the KV transfer are left to the disaggregated stack. We validate this on a three-region testbed of 7 prefill and 3 decode RTX 5090

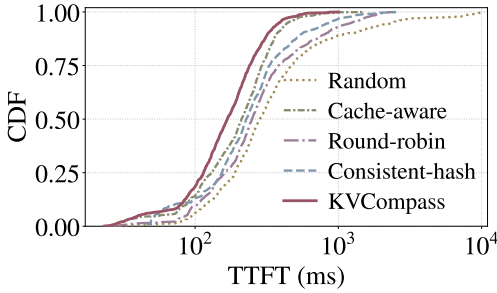


Fig. 18. TTFT CDF under prefill-decode disaggregated serving (Mixed workload).

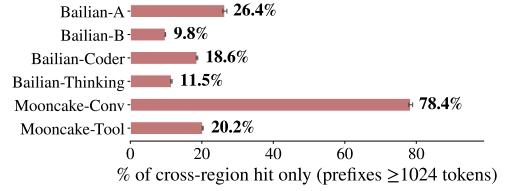


Fig. 19. Cross-region only KV cache hits of ≥ 1024 -token prefixes in production traces from Qwen-Bailian [45] and Mooncake [36].

instances, where the edge node plays SGLang’s prefill-decode load balancer with Mooncake KV transfer [36]. Every scheme selects the prefill worker, and the decode instance is drawn at random within the prefill’s region identically for all schemes, so all separation comes from prefill routing. Figure 18 shows the TTFT CDF at 24 req/s on the hot-skew Mixed workload against the stock prefill-decode stack with random dispatch, SGLang’s built-in cache-aware router, round-robin, and consistent hashing. KVCompass has the lowest median TTFT, 1.25 to 1.69 $\times$ below every baseline, and its P99 is 1.3 to 15.7 $\times$ below, because random and hash dispatch let hot-prefix requests concentrate and their tails stretch past 9 s.

D Cross-Region Reuse in Production Traces

Section 2.1 measures the cross-region reuse opportunity on WildChat. Here we repeat the analysis on production KV cache traces from two other providers, to show the opportunity is not specific to one workload. We replay four Qwen-Bailian usage traces [45] with chat, coder, and reasoning traffic, and the Mooncake conversation and tool-agent traces [36]. These traces carry no user locations. We assign each root session to one of 16 North-American regions in proportion to population, and follow-up turns inherit its region. We replay first-turn requests in arrival order and classify each reusable prefix of at least 1024 tokens as cold, cached in the local region, or cached only in another region, over 20 seeds. Figure 19 reports the remote-only share of hits. Across the six traces, 10 to 78% of reusable prefixes are cached only in another region when a request arrives, and 96 to 100% of shared-prefix reuse volume involves users from more than one region. These additional traces support cross-region reuse opportunities under the assigned region model, rather than measure actual cross-region traffic from these providers.

Received 10 July 2026; revised 5 October 2026; accepted 9 October 2026