University of Houston
Dept. Computer Science
Cloud Computing Project Report 2025

Distributed Inference Key-value Cache in Cloud Setting

Author: Minh Nguyen, Zhenghui Gui

Supervisor: Larry Shi

Abstract

Distributed Inference Key-value Cache in Cloud Setting

Author: Minh Nguyen, Zhenghui Gui

Large language model (LLM) inference typically relies on attention key-value (KV) caches to avoid recomputing past tokens. However, in distributed serving environments where concurrent requests are load-balanced across multiple workers, the same sequence context (e.g., continuation prompts, multi-turn conversations) may be processed by different workers, forcing redundant prefill computation and wasted memory. This work introduces a cloud-native distributed KV cache system that eliminates this redundancy by deterministically routing sequences to dedicated workers using consistent hashing with virtual nodes. The system features a Gateway for request routing, a Coordinator for worker discovery, and GPU Worker nodes that perform inference with persistent KV cache storage, deployed on Google Kubernetes Engine (GKE) with auto-scaling, isolation, and LRU-based memory management. Using GPT-2 inference as a demonstration model, we validate the architecture's ability to maintain cache consistency across workers and improve resource utilization in cloud-based environments. The results highlight the distributed KV cache system as a practical building block for efficient, scalable LLM serving in cloud environments.

Acknowledgements

I would like to thank my TA, my friends, and all my wonderful classmates for their unswerving support during my project. Without your help none of this would have been possible.

I love you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large Language Models (LLMs) have become the backbone of modern AI applications, powering a wide range of applications, from conversational agents, code assistants, to multi-modal reasoning systems, etc. While model architectures and hardware continue to advance, the cost of autoregressive inference remains a fundamental bottleneck. Each generated token requires attending over the entire sequence history, causing latency to grow with context length. As context windows scale to tens or hundreds of thousands of tokens, single-GPU inference becomes increasingly memory-bound and costly. Modern server architectures mitigate this by storing previously computed *key-value (KV) states* as a cache, enabling constant-time retrieval rather than recomputation.

While KV caching significantly improves single-model performance, its benefits degrade in *distributed cloud environments*. Production systems rely on *horizontal scaling*, where multiple GPU workers process concurrent requests. Because traditional serving architectures treat each GPU worker as an isolated execution unit, KV states remain *local* to the worker that originally handled the request. If subsequent requests from the same conversation are routed to a different worker — due to request bursts, auto-scaling events, or load balancing — the new worker must *recompute all prior attention history*, causing:

- Increased latency for multi-turn interactions
- Lower throughput under high concurrency
- Wasted computation and GPU memory
- Reduced scalability despite elastic cloud resources

Although the cloud offers abundant distributed computing, current LLM serving frameworks lack a mechanism for *global discovery and reuse of KV cache across nodes*. These limitations motivate the central research question of this work:

> **How can we enable efficient distributed LLM inference by sharing KV caches across multiple workers with consistent lookup to reduce redundant computation and improve scalability?**

To address this question, we propose a *cloud-native distributed KV cache system* that enables cross-worker reuse of KV states during inference. Instead of splitting a single sequence across multiple nodes, our design assigns each sequence entirely to a single worker but distributes *different sequences* across the cluster using *consistent hashing [LV14]*. This ensures balanced cache utilization and simplifies retrieval semantics.

When a follow-up request arrives, the system performs a *global lookup* through a **Coordinator** service to determine which worker the KV states are stored in. If found, the request is routed or proxied to the correct GPU node to *resume inference without recomputation*, reducing redundant compute and improving responsiveness.

We deploy the full system on *Google Kubernetes Engine (GKE)* using private networking, CPU and GPU node pools, Horizontal Pod Autoscaler (HPA), and pod-level security policies. This design supports multi-session serving, dynamic workload distribution, and scalable long-context inference beyond single-GPU limits. The system includes:

- **Gateway** — public request entry point with *Load Balancer*
- **Coordinator** — sequence-to-node mapping and global cache state
- **GPU Worker Pods** — model execution and KV store with LRU eviction

Our key contributions are summarized as follows:

- **A distributed KV cache system enabling cross-worker reuse.** We design a cloud-native inference architecture that preserves KV cache locality for each sequence while enabling global discovery and reuse across workers, eliminating redundant KV recomputation.
- **Consistent hashing–based cache placement with a lightweight Coordinator.** Our system manages sequence ownership, routing metadata, and dynamic worker membership using consistent hashing, ensuring balanced cache distribution and robustness under auto-scaling.
- **End-to-end serving system with production-aligned microservices.** We implement a modular Gateway–Coordinator–Worker pipeline supporting resumed inference, efficient tensor serialization, and LRU-based memory management in GPU workers.
- **Cloud-supported scalability and improved concurrency efficiency.** Our GKE deployment demonstrates reduced inference latency and higher throughput under concurrent multi-session workloads by reusing previously cached KV states across worker boundaries.

Contributions All source code, deployment automation scripts, and experiment configurations are publicly available in our GitHub repository [NG25].

Chapter 2

Related Work

Operator- and Kernel-Level LLM Optimizations. Recent works such as FlashAttention [DFE+22] and FlashInfer [YCL+25] accelerate attention computation on a single GPU by reducing memory movement and fusing CUDA kernels. These optimizations substantially lower per-token latency, but they remain fundamentally single-node. They do not alleviate memory constraints nor address cross-node KV reuse, both of which are critical in large-scale, cloud-based LLM serving.

Single-Node KV-Cache Management. Systems like vLLM [KLZ+23] introduce mechanisms such as PagedAttention to improve KV-cache residency and batching throughput, enabling efficient reuse of attention states within a single machine. However, KV-cache management remains strictly local, limiting scalability for long-context workloads or high-concurrency serving. Our work extends this line of research by enabling distributed KV caching across multiple GPU workers while preserving locality via consistent hashing.

Distributed LLM Serving and Cluster Scheduling. Frameworks such as Helix [MZM+25] investigate scheduling and model placement across heterogeneous GPU clusters, optimizing for overall throughput and latency. While effective for distributed serving, these systems assume KV tensors remain node-resident and do not attempt to shard or fetch KV across workers. In contrast, our design treats cross-node KV distribution as a first-class architectural primitive, enabling scalable, memory-efficient multi-node LLM inference in cloud environments.

Consistent Hashing Algorithm. Consistent hashing, originally proposed by Karger et al. [KLL+97], provides a distributed hash table mechanism that minimizes key redistribution when nodes are added or removed. Traditional implementations use a hash ring with virtual nodes to achieve better load balancing. Lamping and Veach introduced a stateless, memory-efficient algorithm [LV14] that computes bucket assignments in $O(\log n)$ time without maintaining a ring structure. Our system adopts the virtual node approach on top of standard hash functions, prioritizing simplicity over the minimal memory footprint of jump hash, while achieving the core benefit of consistent hashing: deterministic routing that ensures the same sequence identifier consistently maps to the same worker for KV cache reuse.

Chapter 3

System Design

This chapter describes the design of our cloud-native distributed KV-cache inference system. Our objective is to support efficient autoregressive decoding across multiple GPU workers while preserving KV locality, minimizing redundant computation, and scaling gracefully under high concurrency. We detail the system architecture, cache routing mechanism, GPU-resident KV storage, memory management, asynchronous execution pipeline, and design rationales.

3.1 System Overview

Our design follows a key principle: *a sequence should be logically "owned" by exactly one worker at any point in time*. This ensures that all cached keys and values remain collocated, eliminating cross-node KV synchronization and avoiding recomputation during decoding.

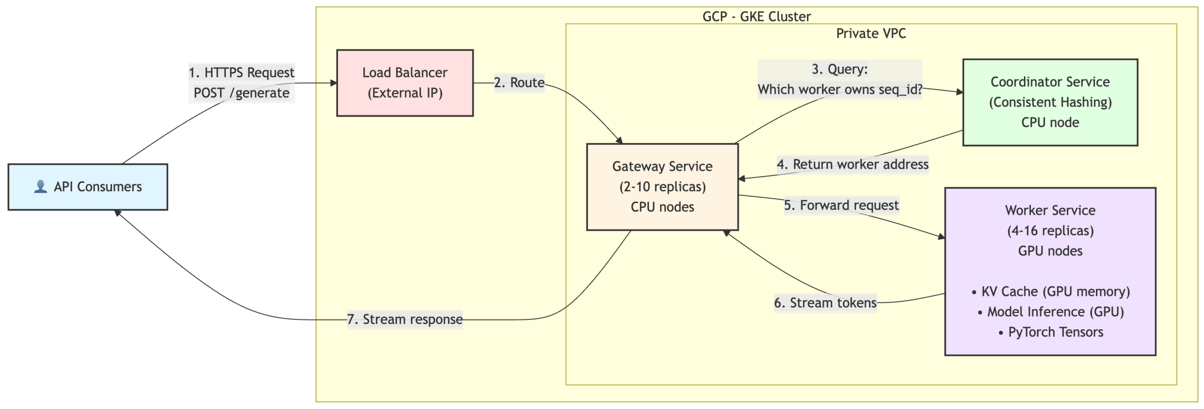

Figure 3.1: System Overview with Gateway, Coordinator, and Worker services, deployed in GKE Cluster

The system is composed of three core microservices:

- **Gateway** — the stateless entry point for inference requests. It exposes a unified API to clients, streams generated tokens, and forwards requests to the correct worker based on coordinator routing.
- **Coordinator** — maintains a consistent hashing ring for sequence-to-worker placement and exposes lightweight routing metadata for every request. It also monitors worker availability to preserve routing consistency under cluster scaling.

- **GPU Workers** — execute transformer forward passes and store sequence KV-cache in GPU memory. Workers append newly generated key-values, manage GPU memory with LRU eviction, and return next-token logits.

All services run within a private GKE VPC (Virtual Private Cloud), and internal communication occurs over lightweight HTTP requests for low-latency control-plane operations. Gateway and Coordinator are stateless, so they can also scale horizontally without requiring data replication or synchronization.

3.2 End-to-End Inference Flow

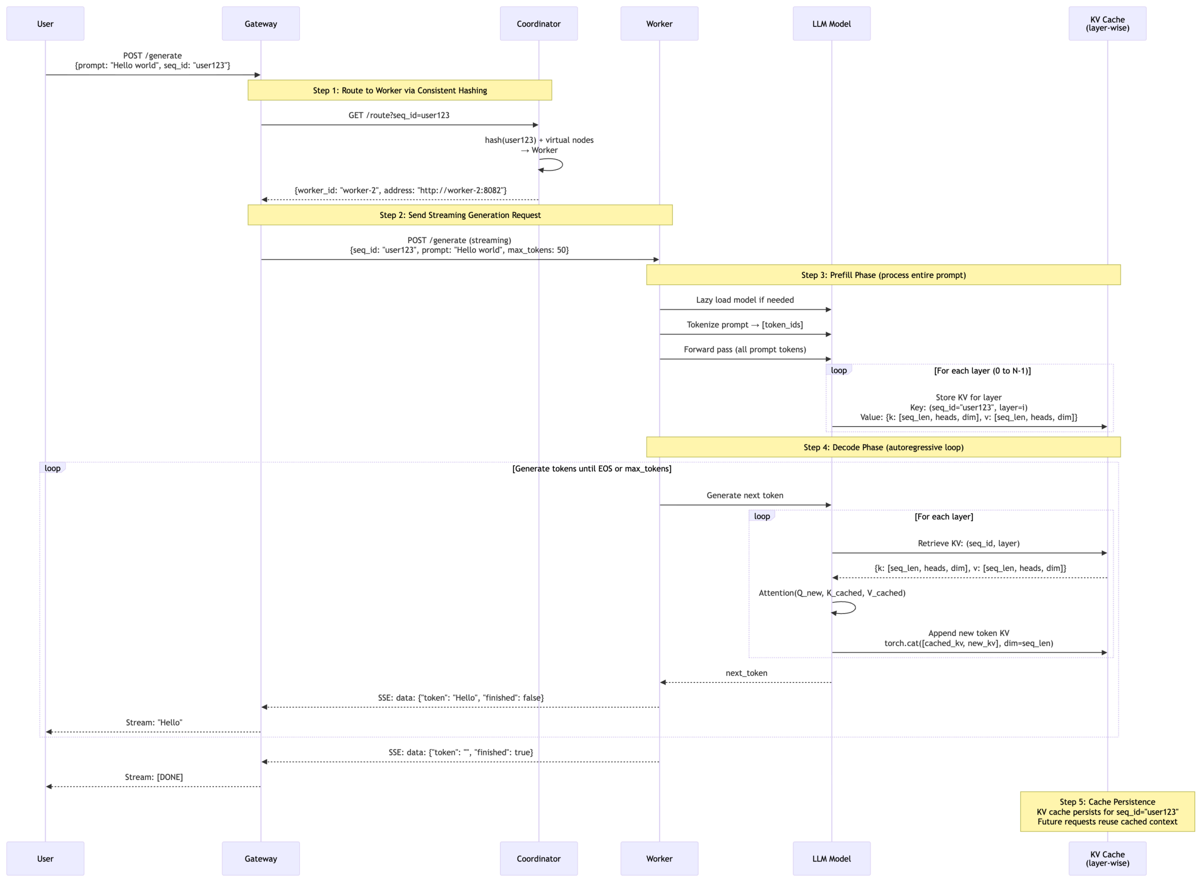
User
Gateway
Coordinator
Worker
LLM Model
KV Cache (layer-wise)
POST /generate {prompt: "Hello world", seq_id: "user123"}
Step 1: Route to Worker via Consistent Hashing
GET /route?seq_id=user123
hash(user123) + virtual nodes → Worker
{worker_id: "worker-2", address: "http://worker-2:8082"}
Step 2: Send Streaming Generation Request
POST /generate (streaming) {seq_id: "user123", prompt: "Hello world", max_tokens: 50}
Step 3: Prefill Phase (process entire prompt)
Lazy load model if needed
Tokenize prompt → [token_ids]
Forward pass (all prompt tokens)
loop
[For each layer (0 to N-1)]
Store KV for layer Key: (seq_id="user123", layer=i) Value: {k: [seq_len, heads, dim], v: [seq_len, heads, dim]}
Step 4: Decode Phase (autoregressive loop)
loop
[Generate tokens until EOS or max_tokens]
Generate next token
loop
[For each layer]
Retrieve KV: (seq_id, layer)
{k: [seq_len, heads, dim], v: [seq_len, heads, dim]}
Attention(Q_new, K_cached, V_cached)
Append new token KV torch.cat([cached_kv, new_kv], dim=seq_len)
next_token
SSE: data: {"token": "Hello", "finished": false}
Stream: "Hello"
SSE: data: {"token": "", "finished": true}
Stream: [DONE]
Step 5: Cache Persistence KV cache persists for seq_id="user123" Future requests reuse cached context
User
Gateway
Coordinator
Worker
LLM Model
KV Cache (layer-wise)

Figure 3.2: End-to-End Token Generation Flow

Figure 3.2 illustrates the interaction among services during a complete autoregressive inference session. The workflow proceeds as follows:

1. **Request routing.** The client sends a generation request to the Gateway, which queries the Coordinator to map the sequence ID to a responsible worker using consistent hashing. The Gateway then forwards all future operations for this sequence to that worker.

2. **Prefill phase.** The worker tokenizes the input prompt and performs a single forward pass over all prompt tokens. All attention K/V tensors produced at each transformer layer are stored in the worker's GPU-resident KV cache. No token is streamed yet during this stage.

3. **Decode phase (autoregressive loop).** For each newly generated token:
   (a) The worker retrieves cached K/V tensors for the sequence.
   (b) The transformer computes the next token using attention over the cached context.
   (c) Newly produced K/V tensors are appended to the existing cache.
   (d) The token is streamed back to the client via the Gateway.

   This continues until EOS or the requested token limit is reached.

4. **Cache retention and reuse.** The KV cache remains stored on the owner worker, enabling future extension of the same sequence without recomputing earlier attention states.

5. **Memory management.** If GPU memory pressure is detected, the eviction policy (LRU) is triggered, while preserving routing correctness through the Coordinator.

This execution pipeline ensures that each sequence's KV state remains fully collocated, eliminating cross-node synchronization and preventing redundant computation during long-context decoding.

3.3 Consistent Hashing with Virtual Nodes

To ensure KV locality and stable assignment under dynamic scaling, we adopt an MD5-based consistent hashing scheme. Let $h(\cdot)$ denote the MD5 hash function mapping sequence identifiers to a 32-bit key space:

$$x = h(\texttt{seq\_id}) \in [0, 2^{32} - 1].$$

Each worker is assigned one or more positions on a virtual hash ring (Figure 3.3):

$$\mathcal{W} = \{w_1, w_2, \ldots, w_{Nk}\},$$

where k is the number of virtual replicas per physical worker to achieve a balanced distribution.

Sequence placement is determined by a clockwise search on the ring:

$$\text{owner}(\texttt{seq\_id}) = \text{next\_clockwise}(x, \mathcal{W}).$$

This yields three desirable properties:

1. **KV Locality (No Sharding Across Workers).** All K/V tensors for a sequence map to exactly one worker, enabling reuse without coordination or inter-node communication during decoding.

2. **Placement Stability with Elastic Scaling.** When workers join or leave (e.g., autoscaling events), only the key-space segment near the affected worker is reassigned:

   $$\text{reshuffle fraction} = O(1/N).$$

   This avoids large-scale cache invalidation or migration.

3. **Smooth Load Distribution.** Virtual replica nodes ensure near-uniform mapping of active sequences even under skewed workloads.

This routing mechanism is fully stateless at the Coordinator: no session tracking is required beyond knowledge of the active hash ring. This allows tens of thousands of independent sequences to be dispatched efficiently and consistently.

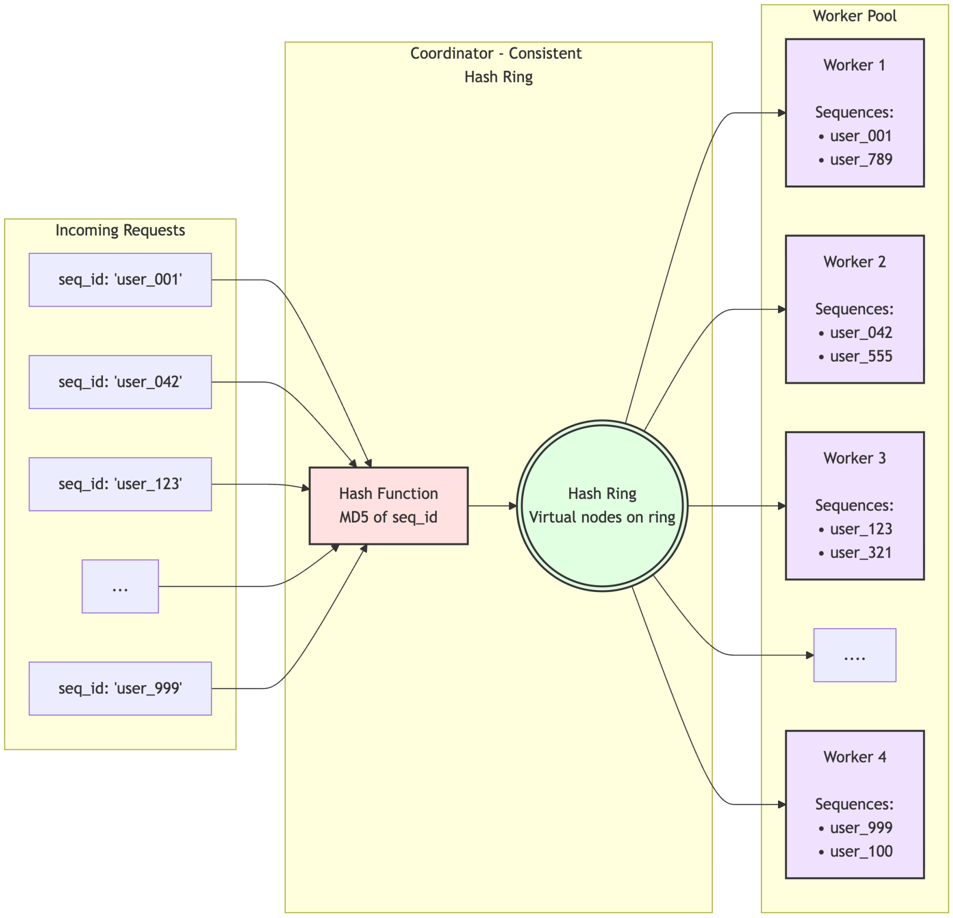
Incoming Requests
seq_id: 'user_001'
seq_id: 'user_042'
seq_id: 'user_123'
...
seq_id: 'user_999'
Coordinator - Consistent Hash Ring
Hash Function MD5 of seq_id
Hash Ring Virtual nodes on ring
Worker Pool
Worker 1 Sequences: • user_001 • user_789
Worker 2 Sequences: • user_042 • user_555
Worker 3 Sequences: • user_123 • user_321
....
Worker 4 Sequences: • user_999 • user_100

Figure 3.3: Consistent Hashing with Virtual Nodes Architecture used in Coordinator for balanced sequence-to-worker mapping

3.4 Worker Internal Architecture

Each worker in our distributed inference system operates as an autonomous compute-and-cache unit responsible for serving all tokens of its assigned sequences. To maximize throughput and minimize latency, workers maintain KV caches directly on the GPU, execute token generation in asynchronous micro-batches, and implement memory-aware eviction to ensure bounded GPU utilization. This design (Figure 3.4) enables high locality, low coordination overhead, and scalable performance as workers are dynamically added or removed. The following subsections detail the key components of the worker architecture.

3.4.1 GPU-Resident KV-Cache Storage

Each worker maintains a KV dictionary stored entirely in GPU memory:

$$(\text{seq\_id}, \ell) \rightarrow \{K_\ell, V_\ell\}.$$

For a transformer with H heads and head dimension D, the cached tensors take the form:

$$K_\ell \in \mathbb{R}^{L\times H\times D}, \qquad V_\ell \in \mathbb{R}^{L\times H\times D},$$

where L is the current sequence length.

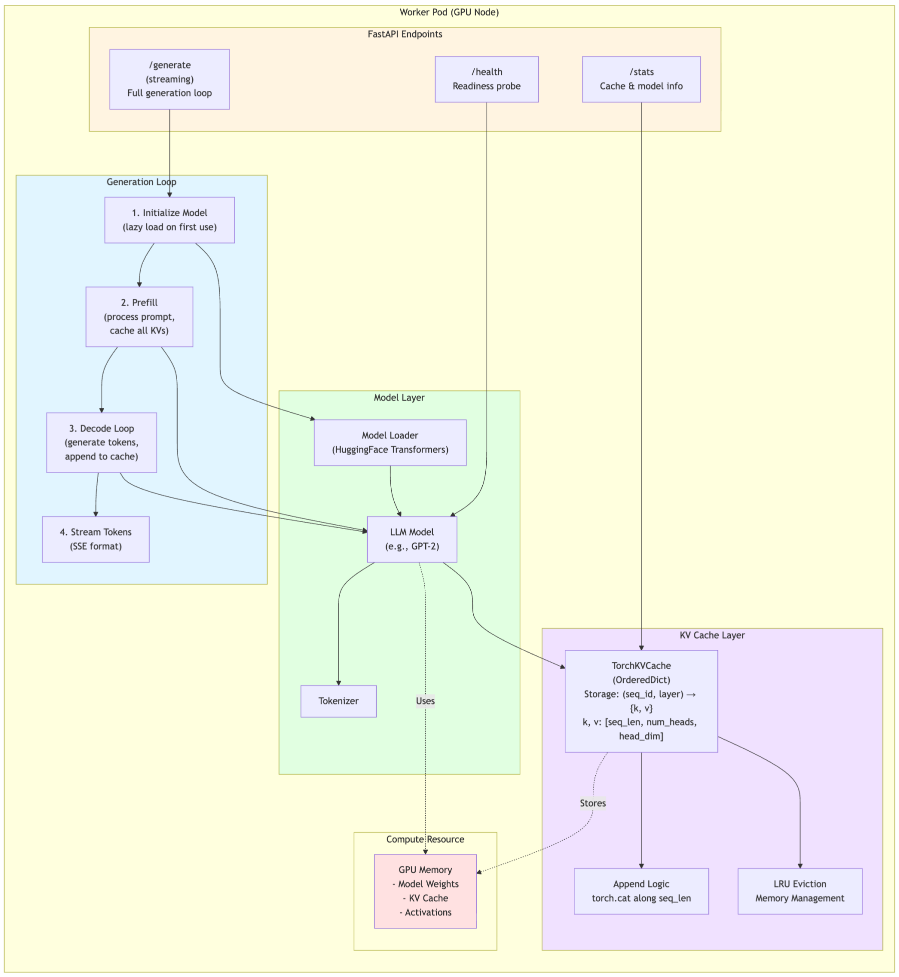
Worker Pod (GPU Node)
FastAPI Endpoints
/generate (streaming) Full generation loop
/health Readiness probe
/stats Cache & model info
Generation Loop
1. Initialize Model (lazy load on first use)
2. Prefill (process prompt, cache all KVs)
3. Decode Loop (generate tokens, append to cache)
4. Stream Tokens (SSE format)
Model Layer
Model Loader (HuggingFace Transformers)
LLM Model (e.g., GPT-2)
Tokenizer
Uses
KV Cache Layer
TorchKVCache (OrderedDict) Storage: (seq_id, layer) → {k, v} k, v: [seq_len, num_heads, head_dim]
Stores
Compute Resource
GPU Memory - Model Weights - KV Cache - Activations
Append Logic torch.cat along seq_len
LRU Eviction Memory Management

Figure 3.4: GPU Worker Node Internal Architecture

When a new token t is generated, the worker computes:

$$(k_t, v_t) = \text{AttentionProj}(x_t),$$

and updates:

$$K_\ell^{(t+1)} = \text{concat}(K_\ell^{(t)}, k_t), \quad V_\ell^{(t+1)} = \text{concat}(V_\ell^{(t)}, v_t).$$

This design offers several benefits:

- **Zero-Copy Access**: All KV tensors remain on GPU, enabling sub-millisecond lookup.
- **No Cross-Node KV Sync**: Sequences never migrate during decoding.
- **Multi-Sequence Parallelism**: Workers can handle hundreds of sequences concurrently, leveraging GPU batching.

3.4.2 KV Retrieval and Compute Pipeline

Given a token index t for sequence s, the worker executes:

1. Retrieve $\{K_\ell, V_\ell\}$ for all layers ℓ.

2. Compute attention using:

$$\text{Attention}(Q_t, K_\ell, V_\ell) = \text{softmax}\left(\frac{Q_t K_\ell^\top}{\sqrt{d}}\right) V_\ell.$$

3. Pass through MLP and residual connections.

4. Produce the next-token logits.

Because KV states grow linearly with L, this design avoids recomputing projections for all previous tokens, saving $O(L)$ computation per step.

3.4.3 Memory-Aware Eviction

GPU memory is finite, and KV tensors accumulate over long sequences. We maintain a sequence-level LRU structure:

$$\text{LRU}(s) = \text{LastAccessStep}(s).$$

When memory usage exceeds threshold τ, we evict:

$$s^* = \arg\min_s \text{LRU}(s),$$

removing its KV tensors across all layers.

However, LRU's assumption of temporal locality often fails in distributed settings. The importance of a token may be global, while LRU only observes local access patterns. This motivates future extensions like attention-aware eviction, where workers track:

$$I_{\text{local}}(s) = \sum_\ell \|\alpha_\ell^{(t)}\|_1,$$

and forward summary statistics to the coordinator.

3.4.4 Asynchronous Batched Token Generation

Workers process decoding steps asynchronously, executing multiple sequences in parallel. The runtime maintains a dynamic batch B_t of active sequences:

$$B_t = \{s_1, s_2, \ldots, s_m\}.$$

Decoding proceeds in three phases:

1. **KV Read**: Gather tensors for all sequences in B_t.

2. **Batched Forward Pass**: Execute the transformer on the batched queries:

$$Y_t = \text{Decoder}(X_t, K, V).$$

3. **KV Write**: Update KV tensors for each sequence.

Batch size m is dynamically adjusted based on GPU utilization (targeting 70% for the best throughput-latency tradeoff).

3.5 Design Rationale & Trade-offs

Why Consistent Hashing? Alternative strategies like round-robin routing or centralized schedulers incur frequent KV migrations or require global locks under scaling events. Consistent hashing avoids both problems, ensures locality and stability, but at the cost of potential imbalance — it is mitigated via virtual nodes.

Why GPU-Resident KV? Storing KV states in CPU memory or external KV stores reduces GPU memory pressure but requires expensive PCIe or network transfers each token step. GPU-resident storage significantly lowers latency, though it introduces tight memory budgets and necessitates eviction policies.

Why Sequence-Level Eviction? Token-level eviction maximizes utilization granularity but risks breaking attention continuity and increases lookup complexity. Sequence-level eviction favors correctness and simplicity at the cost of occasionally freeing more memory than strictly necessary. This fits natural LLM usage patterns.

Why Asynchronous Batching Execution? Synchronous decoding simplifies scheduling but under-utilizes GPU compute, especially with many active short sequences. Asynchronous micro-batching improves arithmetic intensity and throughput, though it introduces batching delay and more complex worker logic.

Chapter 4

Implementation and GKE Deployment

4.1 Implementation Overview

The system is implemented in *Python 3.13+* using *FastAPI* for asynchronous HTTP request handling and *Server-Sent Events (SSE)* for token-by-token streaming. The machine learning stack is built on *PyTorch* and *HuggingFace Transformers*, supporting pre-trained decoder-only models (e.g., GPT-2) and aligning with modern LLM serving pipelines.

Core Components. The Gateway acts as the public entry point, applying request validation and batching before forwarding inference requests. The Coordinator provides deterministic routing via consistent hashing with 100 virtual nodes per worker, ensuring balanced distribution and minimal reshuffling when scaling. Worker processes maintain *GPU-resident* KV caches, indexed by `(seq_id, layer_index)` with tensor shape `[seq_len, num_heads, head_dim]`. During autoregressive decoding, Workers retrieve existing K/V entries, perform a single forward pass for the next token, append new tensors, and stream output tokens. Cache eviction uses an LRU strategy instrumented through `OrderedDict`, triggered when GPU memory pressure is detected.

Technology Stack.

- **Application Layer**: FastAPI + Uvicorn, SSE for interactive streaming
- **Model Runtime**: PyTorch 2.5+, Transformers 4.57+
- **Containerization**: Docker with GPU-enabled CUDA base images
- **Orchestration**: Kubernetes Deployments (stateless services) and StatefulSets (Workers)
- **Provisioning**: Terraform for reproducible GKE infrastructure

Design Considerations. Python enables seamless integration with the model runtime while supporting non-blocking streaming execution. KV caches remain in GPU memory to avoid PCIe transfer overhead and preserve the performance benefit of past-context reuse. Consistent hashing avoids centralized scheduling bottlenecks and ensures low-overhead failover and scaling. The system is intentionally stateless at the routing layer; only Workers maintain model state, which simplifies autoscaling and recovery procedures.

Source code The full implementation, including FastAPI services, Kubernetes manifests, and Terraform automation, is available in our open-source repository [NG25].

4.2 Component Implementation

4.2.1 Gateway

The Gateway exposes the `/generate` API endpoint for streaming inference. Upon receiving a request identified by `seq_id`, the Gateway queries the Coordinator via `GET /route/<seq_id>` to determine the responsible worker. It then establishes a streaming HTTP connection to the worker and forwards the response to the client via Server-Sent Events (SSE). This decoupling of routing and streaming enables the Gateway to scale independently, supporting high concurrency and minimizing head-of-line blocking.

4.2.2 Coordinator

The Coordinator maintains a registry of active workers and performs consistent-hash routing. Workers register on startup through `POST /register` with their identifier and endpoint, and deregister via `POST /deregister` during graceful shutdown. For each worker `worker_id`, the hash ring stores 100 virtual nodes generated by hashing `"<worker_id>-vn-<i>"` for `i in [0, 100)`. For a given `seq_id`, the Coordinator hashes the identifier and selects the next virtual node clockwise on the ring, returning the corresponding physical worker. This ensures stable KV locality while limiting remapping to `O(1/N)` sequences under scaling events.

4.2.3 Worker

Each Worker node implements the complete inference pipeline within a single `/generate` endpoint. The process consists of three phases:

1. **Initialization.** On the first request, the Worker lazily loads the specified language model and tokenizer into memory (GPU or CPU based on availability).

2. **Prefill (Context Encoding).** The Worker tokenizes the prompt and performs a single forward pass across all transformer layers. Attention outputs produce key and value tensors of shape `[seq_len, num_heads, head_dim]`, which are stored in a GPU-resident cache indexed by `(seq_id, layer_index)`.

3. **Decode.** The Worker enters an autoregressive generation loop. At each step, for every layer, the Worker retrieves the cached KV tensors, computes attention with the new query token, and appends the resulting KV pair to the cache using `torch.cat([cached_kv, new_kv], dim=0)`, incrementally growing the sequence length dimension. Generated tokens are streamed to the Gateway in real-time via SSE.

Compute and streaming operations are asynchronous to maintain high GPU utilization. When memory pressure is detected, LRU-based eviction removes the least recently active sequences while preserving locality guarantees.

4.3 GKE Deployment Architecture

We deploy our distributed inference system on *Google Kubernetes Engine (GKE)* to support elastic scaling, operational isolation, and low-latency communication within a private cluster. Figure 4.1 illustrates the deployment topology.

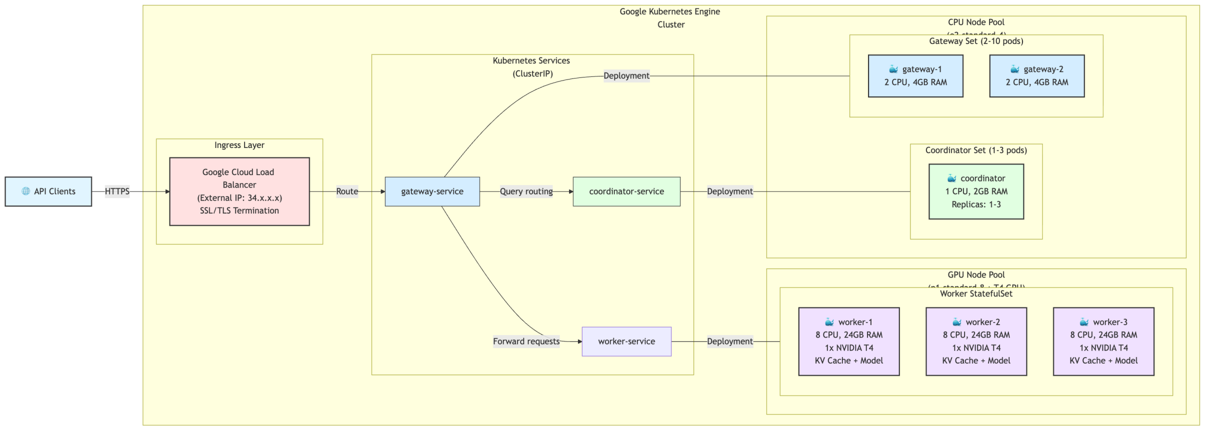
Google Kubernetes Engine Cluster
API Clients
HTTPS
Ingress Layer
Google Cloud Load Balancer (External IP: 34.x.x.x) SSL/TLS Termination
Route
Kubernetes Services (ClusterIP)
gateway-service
Deployment
Query routing
coordinator-service
Deployment
Forward requests
worker-service
Deployment
CPU Node Pool
Gateway Set (2-10 pods)
gateway-1 2 CPU, 4GB RAM
gateway-2 2 CPU, 4GB RAM
Coordinator Set (1-3 pods)
coordinator 1 CPU, 2GB RAM Replicas: 1-3
GPU Node Pool
Worker StatefulSet
worker-1 8 CPU, 24GB RAM 1x NVIDIA T4 KV Cache + Model
worker-2 8 CPU, 24GB RAM 1x NVIDIA T4 KV Cache + Model
worker-3 8 CPU, 24GB RAM 1x NVIDIA T4 KV Cache + Model

Figure 4.1: Google Kubernetes Engine (GKE) Deployment Plan

To balance cost efficiency and performance during development, we provision a *zonal cluster*. This avoids the higher resource redundancy and pricing of *regional clusters*, while maintaining full autoscaling capabilities. In a production setting where cross-zone resilience and multi-regional user latency are critical, the same deployment can be migrated to a *regional cluster* with no architectural changes.

The cluster consists of two dedicated node pools tailored to workload type:

- **CPU Node Pool**: cost-efficient machines (e.g., `e2-standard-4`), running Gateway and Coordinator services that handle routing and HTTP streaming with minimal compute load.
- **GPU Node Pool**: NVIDIA T4–equipped instances (e.g., `n1-standard-8 + T4`), running the Worker `StatefulSet` responsible for transformer inference and GPU-resident KV caching.

Each logical service is containerized and stored in *Google Artifact Registry* (Table 4.1):

Table 4.1: Deployment container images for each service.

Component	**Container Image**	**Runtime**
Gateway	`REGION-docker.pkg.dev/.../gateway:latest`	FastAPI + SSE
Coordinator	`REGION-docker.pkg.dev/.../coordinator:latest`	Consistent hashing
Worker	`REGION-docker.pkg.dev/.../worker:latest`	LLM + KV cache

All inter-service communication uses *ClusterIP* Services within the private GKE VPC, ensuring secure and low-latency communication. A *Google Cloud Load Balancer* terminates external HTTPS traffic and forwards requests to the Gateway Service only. Inference requests follow the distributed routing design:

1. Gateway queries the Coordinator for the consistent-hash mapping from `seq_id` to Worker.
2. Gateway issues a streaming generation request to the selected Worker pod.
3. The Worker performs KV-local inference and streams tokens back to the client.

Autoscaling ensures cost efficiency under fluctuating demand:

- **Gateways**: Horizontal Pod Autoscaler (HPA), 2–10 replicas at 70% CPU target.
- **Coordinators**: 1–3 replicas for high availability.
- **Workers**: GPU `StatefulSet` scaled via GKE node autoscaler at 80% GPU utilization.

By isolating CPU-bound and GPU-bound components into separate node pools and leveraging zonal deployment for cost optimization, this architecture provides a scalable, resilient, and production-ready foundation for distributed LLM inference workloads.

4.4 Network & Security Policies

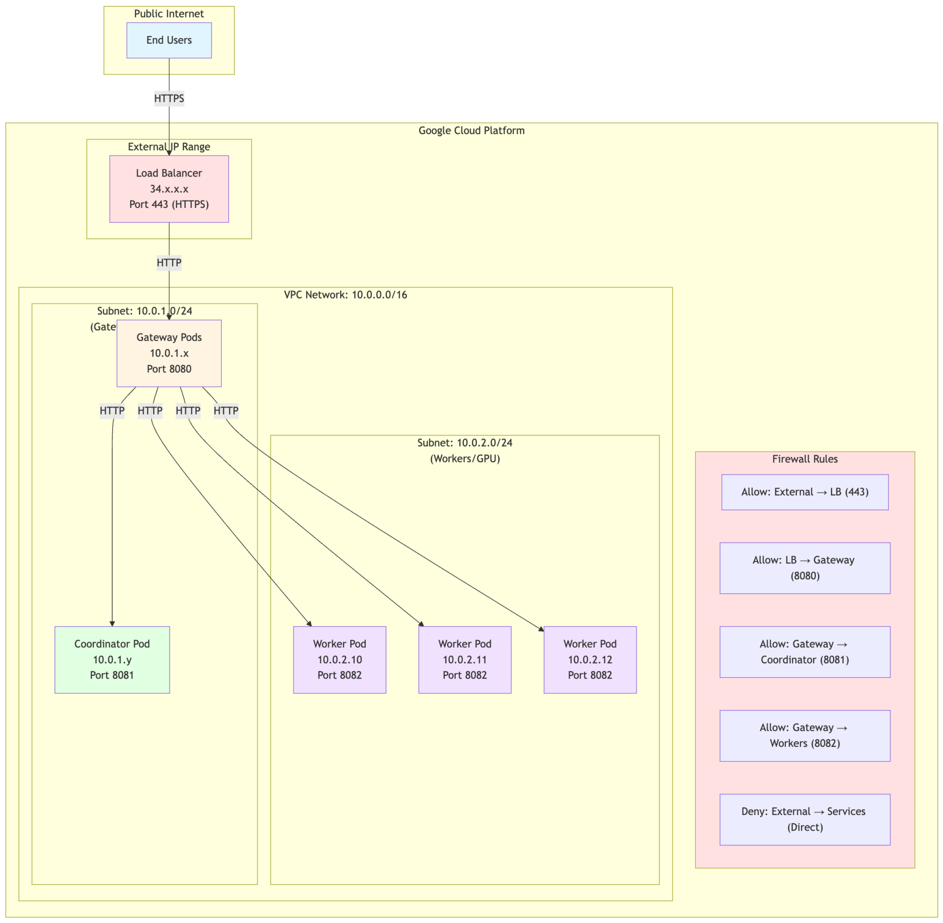

Figure 4.2: System network topology and security boundaries.

The deployment follows a defense-in-depth design with strict network segmentation. The Gateway service is the only component exposed externally, accessed through a Google Cloud Load Balancer with HTTPS termination. All backend services - Coordinator and Workers - communicate exclusively through `ClusterIP` Services, ensuring that no inference or routing endpoints are directly reachable from the public internet.

Network Isolation Each node pool resides within a private Virtual Private Cloud (VPC) with auto-allocated subnets. Firewall rules are configured to permit HTTP (port `80`) and HTTPS (port `443`) traffic exclusively to nodes tagged with the Gateway role. Internal cluster communication utilizes Kubernetes DNS for service discovery, where services resolve to stable ClusterIP addresses (e.g., `coordinator.default.svc.cluster.local`). Worker pods in the StatefulSet receive predictable DNS names (`worker-0.worker.default.svc.cluster.local`), enabling direct addressing for deterministic routing.

Authentication and Authorization Authentication is not enforced in the current proof-of-concept configuration. However, in production deployments, this architecture supports integration with cloud-native authentication mechanisms such as Google Cloud Identity-Aware Proxy (IAP) for the LoadBalancer endpoint, OAuth2-based identity providers, or signed API keys validated at the Gateway level. Within the cluster, Google Kubernetes Engine adopts Workload Identity for pod-to-cloud authorization, ensuring that each microservice receives the least privileges required to access Google Cloud resources.

4.5 Infrastructure as Code

We adopt a fully declarative deployment workflow using Terraform for cloud infrastructure provisioning and Kubernetes manifests for workload orchestration, ensuring reproducibility, security, and versioned configuration.

Terraform Configuration The Terraform configuration provisions a *zonal GKE cluster* in `us-central1-a` along with two workload-specific node pools and their associated networking resources. For development cost efficiency, CPU node pools use `e2-small` and `e2-standard-4` machine types; however, the configuration supports seamless upgrades to production-grade instances (e.g., `n1-standard-8` with NVIDIA T4 GPUs) by modifying variables without changing cluster topology. Terraform also manages VPC creation, service accounts, and firewall policies, including HTTPS access for the Gateway Load Balancer. Infrastructure state is stored remotely to support collaborative changes and rollbacks.

Containerization Each system component is packaged as a Docker image using multi-stage builds. Gateway and Coordinator containers use a Python 3.13 slim base with FastAPI dependencies installed via the `uv` package manager, minimizing image size. Worker images bundle PyTorch and HuggingFace Transformers, with optional CUDA-enabled variants for GPU acceleration. All images are tagged via semantic versioning or Git commit hashes and pushed to *Google Artifact Registry* for controlled rollout.

Kubernetes Manifests Workloads are deployed following cloud-native best practices: stateless routing components (Gateway, Coordinator) run as `Deployments` to support rolling updates, while Workers run as a `StatefulSet` to preserve stable identities (e.g., `worker-0`, `worker-1`) crucial for deterministic sequence routing and KV cache locality. Worker pods are internally routable via per-pod `ClusterIP` Services, while the Gateway is exposed externally using a managed `LoadBalancer` Service.

Deployment Automation Deployment is fully automated through a set of shell scripts that integrate Terraform provisioning and Kubernetes rollout. The `build_images.sh` script builds service containers and pushes them to Google Artifact Registry, creating repositories on demand. The `deploy_gke.sh` script configures cluster credentials, substitutes environment-specific values (project ID, region, zone, image tags), and applies Kubernetes manifests. For end-to-end deployments, `quickstart_gke.sh` automates the entire process: enabling required GCP APIs, provisioning infrastructure via Terraform, building and publishing container images, and deploying workloads to the cluster. This automation ensures repeatable, environment-agnostic deployments with minimal manual intervention.

Chapter 5

Experimental Results

5.1 Experimental Setup

All experiments were conducted on a local development environment using a MacBook with Apple Silicon (M-series processor). he system was implemented in *Python 3.13* with *PyTorch 2.5+* and *HuggingFace Transformers 4.57+*. *GPT-2* served as the reference language model for all tests. Due to the lack of CUDA support on macOS (Apple Silicon does not support NVIDIA CUDA), all experiments were executed on CPU. While this limits absolute performance compared to GPU acceleration, it ensures reproducibility across different hardware configurations and does not affect the validity of comparative results (e.g., cache vs. no-cache performance). Tests utilized the *pytest* framework with custom performance metrics collection.

5.2 KV Cache Performance Evaluation

We evaluated the effectiveness of KV caching by comparing generation latency with and without cache reuse for continuation requests.

Single Generation Comparison Table 5.1 presents results for a continuation task where the initial prompt ”The future of artificial intelligence” (6 tokens) was cached during a first generation. The subsequent generation extends this context by 15 additional tokens. In the baseline scenario, the worker receives the complete prompt including all prior tokens, simulating a fresh sequence without cache access. In the cached scenario, the worker retrieves the 6-token KV cache and only computes attention for the 15 new tokens.

Table 5.1: Cache vs. No-Cache Performance Comparison

Metric	No Cache	With Cache
Total time	1.264s	0.796s
Time to first token	1.263s	0.795s
Tokens/sec	11.87	18.84
New tokens generated	15	15
Improvement	**-**	**+37.0%**

The cached generation achieves 37% reduction in total time and time to first token (TTFT), with a 58.8% improvement in throughput (tokens/sec). This demonstrates the significant benefit of avoiding redundant prefill computation for continuation requests.

Multi-Round Cache Benefit Table 5.2 presents results from three sequential generations with different prompts, demonstrating consistent cache performance across multiple interactions.

Table 5.2: Multi-Round Cache Performance

Round	Prompt	Total Time	TTFT	Tokens/sec
1	”Artificial intelligence is”	0.942s	0.941s	10.61
2	”Machine learning can”	1.013s	1.012s	9.87
3	”Neural networks are”	0.566s	0.566s	17.66
Average	-	**0.841s**	**0.840s**	**12.71**

The consistent performance across rounds (average TTFT: 0.840s, throughput: 12.71 tokens/sec) validates stable cache utilization and management across multiple sequences.

Sequence Length Impact Table 5.3 compares performance between short and long sequences to assess cache append complexity.

Table 5.3: Performance Across Sequence Lengths

Sequence Length	Total Time	Tokens/sec	Time per Token
Short (10 tokens)	0.512s	19.53	51.2ms
Long (30 tokens)	1.493s	20.09	49.8ms

The consistent per-token latency (approximately 50ms) across sequence lengths validates that KV cache append operations maintain O(1) amortized complexity, as the `torch.cat` operation scales linearly with the number of new tokens, not the total cached sequence length.

5.3 Routing and Cache Locality Validation

The consistent hashing implementation with 100 virtual nodes per worker was validated through comprehensive distribution and locality tests using 5,000 distinct sequences distributed across 4 workers.

Distribution Uniformity Table 5.4 presents results from distributing 5,000 sequences across 4 workers using consistent hashing with virtual nodes.

The distribution demonstrates excellent load balancing with sequence counts ranging from 1,205 to 1,333 per worker. The maximum deviation of 128 sequences represents only 2.56% variance from perfect balance (1,250 sequences per worker), validating the effectiveness of virtual nodes at scale. This near-uniform distribution across 5,000 sequences confirms that the consistent hashing algorithm maintains balance even under significant load.

Table 5.4: Sequence Distribution Across Workers (5,000 sequences)

Worker ID	Sequences Assigned	Distribution
eeece8479907	1,333	26.66%
4ddbfbce674c	1,230	24.60%
0d65be9952ec	1,205	24.10%
88a0a5d2745f	1,232	24.64%
Total	**5,000**	**100%**
Balance Range	**1,205-1,333**	**(128 seq variance)**

Routing Consistency The routing stability test verified that the same sequence identifier consistently maps to the same worker across 20 repeated routing requests. All 20 requests for sequence `'test-seq-123'` correctly routed to worker `4ddbfbce674c`, confirming deterministic routing behavior essential for cache locality.

Cache Locality Verification Table 5.5 presents results from cache locality and append operation tests.

Table 5.5: Cache Locality and Layer-wise Storage Validation

Test Case	Initial State	Final State
Cache Locality Test		
Sequence ID	`locality-test-seq`	
Assigned worker	`bf6469b980b4`	
KV tensor shape	`[5, 12, 64]`	
Cache Append Behavior Test		
Initial sequence length	5 tokens	-
After 3 appends	-	8 tokens
Initial KV shape	`[5, 12, 64]`	-
Final KV shape	-	`[8, 12, 64]`
Multi-layer Cache Test		
Sequence ID	`multi-layer-seq`	
Number of layers	4 (all retrieved successfully)	

The cache locality test confirms that sequences route to the correct worker and KV tensors maintain the expected shape `[seq_len, num_heads, head_dim]` where `num_heads=12` and `head_dim=64` for GPT-2. The append behavior test validates that the sequence length dimension grows correctly (5 $\rightarrow$ 8 tokens) through `torch.cat` operations while preserving the attention head structure. The multi-layer test confirms that all 4 transformer layers store and retrieve KV caches independently under the `(seq_id, layer_index)` key structure.

Gateway Routing Integration The end-to-end routing test verified that the Gateway correctly queries the Coordinator for routing decisions. Test sequence `'gateway-test-seq'` was successfully routed to worker `88a0a5d2745f`, confirming proper integration between Gateway, Coordinator, and Worker components.

These validation tests demonstrate that the consistent hashing implementation achieves:

- Excellent distribution uniformity at scale (2.56% variance for 5,000 sequences across 4 workers)
- Deterministic routing (100% consistency across repeated requests)
- Correct cache locality (sequences always access their designated worker's cache)
- Proper layer-wise storage (independent KV cache entries per transformer layer)
- Correct append semantics (sequence length grows via concatenation)

5.4 System Stress Testing

Table 5.6 summarizes results from concurrency, sustained load, and burst traffic tests.

Table 5.6: Stress Test Results

Test Type	Total Requests	Success Rate	Avg Latency	Throughput
High Concurrency (20 concurrent)	20	100%	8.283s	1.42 req/s
Sustained Load (30s @ 2 req/s)	48	100%	1.099s	1.55 req/s
Burst Traffic (alternating)	40	100%	7.371s	-

All stress tests achieved 100% success rate with no failures, demonstrating robust concurrent request handling. The sustained load test's P95 latency of 2.007s and average throughput of 6.02 tokens/sec reflect aggregate system capacity under continuous load.

High Concurrency We evaluated system behavior under concurrent load by launching 20 simultaneous generation requests. The system achieved 100% success rate (20/20 requests) with an average request latency of 8.283s and aggregate throughput of 1.42 requests/sec. All requests completed successfully without failures, demonstrating robust concurrent request handling.

Sustained Load A 30-second sustained load test at 2 requests/sec target rate processed 48 requests with 100% success rate. Actual throughput measured 1.55 req/sec with average latency of 1.099s and P95 latency of 2.007s. The average throughput of 6.02 tokens/sec reflects the aggregate system capacity under sustained load.

Burst Traffic The system was subjected to alternating bursts of high traffic (15-20 concurrent requests) and quiet periods (2-3 requests). All 40 requests across multiple bursts completed successfully (100% success rate) with average latency of 7.371s, demonstrating resilience to traffic variations.

Table 5.7: Mixed Workload Results

Category	Requests	Success Rate	Avg Time	Avg Tokens
Short sequences	5	100%	8.953s	5.0
Medium sequences	5	100%	9.788s	10.0
Long sequences	5	100%	8.324s	20.0

Mixed Workload A test with varying sequence lengths (5, 10, and 20 tokens) showed the system handles heterogeneous workloads effectively. Short sequences completed in 8.953s (5 tokens avg), medium sequences in 9.788s (10 tokens avg), and long sequences in 8.324s (20 tokens avg), with 100% success rate across all categories.

Table 5.7 shows system behavior under heterogeneous sequence lengths.

The system handles heterogeneous workloads effectively with consistent success rates across all sequence length categories.

5.5 Functional Correctness

The `test_generation_flow.py` suite validated core system functionality:

- **Basic Generation**: Successfully generated 10 tokens in 11.81s (0.85 tokens/sec) with correct token sequence
- **Model Auto-Initialization**: Lazy loading mechanism correctly initialized GPT-2 on first request
- **Temperature Variation**: Generation quality varied appropriately across temperature settings (0.1, 0.7, 1.0)
- **Concurrent Generations**: Successfully handled 3 concurrent requests with different prompts
- **Performance Metrics**: Achieved 17.84 tokens/sec for 20-token generation with 1.120s TTFT and 56.0ms average per-token latency

Table 5.8 summarizes results from the generation flow validation tests.

Table 5.8: Functional Correctness Tests

Test Case	Result	Key Metric
Basic generation (10 tokens)	PASSED	0.85 tokens/sec, 11.81s total
Model auto-initialization	PASSED	GPT-2 lazy load successful
Temperature variation (0.1, 0.7, 1.0)	PASSED	Output diversity verified
Concurrent generations (3 requests)	PASSED	All completed successfully
Performance metrics (20 tokens)	PASSED	17.84 tokens/sec, 1.120s TTFT

All functional tests passed with 100% success rate, validating the correctness of the generation pipeline, streaming implementation via Server-Sent Events, and layer-wise KV cache storage and retrieval operations.

5.6 Overall Results

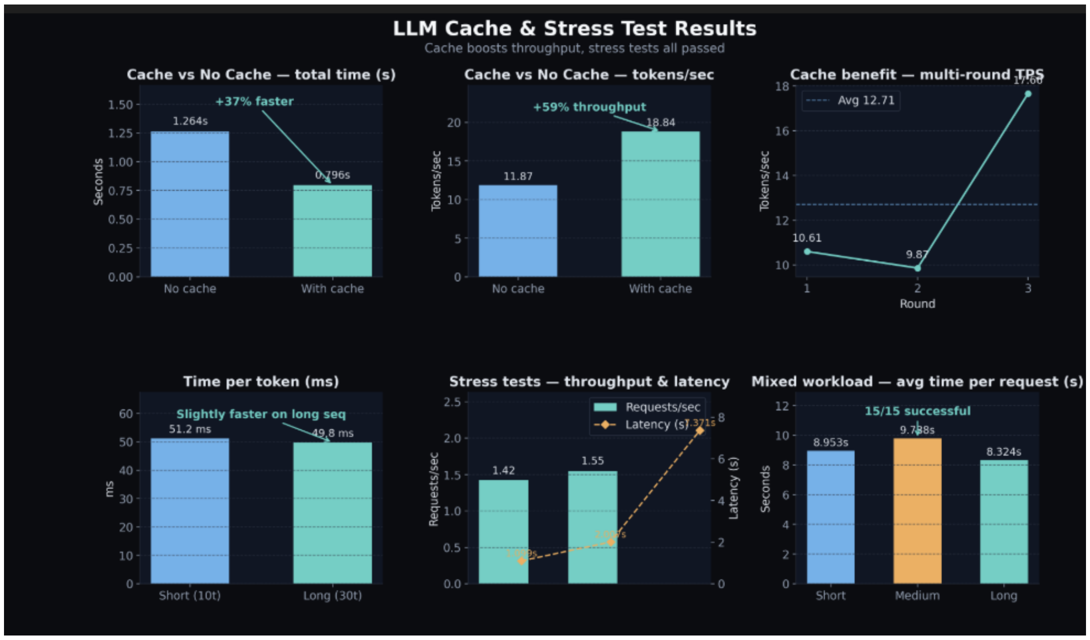

Figure 5.1: Overall Results Comparison

Top-left and top-middle panels report latency and throughput for single-turn generation with caching enabled (overall generation time down 37, tokens/sec up 59). The top-right panel shows average TPS across multi-turn interactions, with a clear jump starting from the third turn. The bottom-left compares time per token for short vs. long sequences, indicating that caching does not introduce degradation on long inputs. The bottom-middle summarizes the request rate and latency under concurrent and sustained load. The bottom-right shows that mixed workloads (short/medium/long prompts) all achieve a 100% success rate.

Chapter 6

Conclusion and Future Work

In this project, we presented a cloud-native distributed key-value (KV) cache architecture for efficient autoregressive LLM inference. The system implements a Gateway–Coordinator–Worker microservice pattern and is deployed on Google Kubernetes Engine (GKE). Key design features include: consistent-hashing–based sequence placement to preserve KV locality, GPU-resident per-sequence KV caches to avoid PCIe/network roundtrips, asynchronous streaming inference via FastAPI and SSE, and a lightweight LRU-based eviction policy in the prototype. The Coordinator provides stable sequence-to-worker mapping even under autoscaling, and the Gateway exposes a single public entry point that forwards streamed generation requests to the appropriate worker.

Our implementation demonstrates that distributing KV state at the sequence granularity enables efficient reuse of attention tensors across follow-up requests, reducing redundant recomputation while supporting elastic scaling across multiple GPU workers. The design separates routing concerns from model execution, simplifies autoscaling, and keeps latency-critical operations colocated with GPU compute.

Despite these benefits, several opportunities remain to improve performance, robustness, and production readiness. We highlight the most promising directions below:

- **Attention-aware eviction policies.** The prototype uses an LRU policy for simplicity. A more effective strategy is to evict entries with the lowest attention importance (salience) under memory pressure. Because attention scores reflect the contribution of cached tokens to current predictions, attention-aware eviction can better preserve model accuracy while freeing memory.

- **Optional persistence and hybrid caching.** Introducing a persistent or tiered cache layer (e.g., etcd, Redis, or object storage) would allow cold KV state to be spilled off GPU memory and restored on demand, increasing effective capacity and reducing cold-start recomputation. A hybrid design can trade latency for capacity in a controlled way.

- **Workload-aware placement and load balancing.** Enhancing the Coordinator with workload signals (traffic patterns, session length, or historical hit rates) can improve placement decisions beyond pure hashing, further reducing cross-node transfers and hotspots.

- **Multi-region and multi-cluster deployment.** Extending the system for regional or global deployments would improve availability and user-perceived latency across geographies. This requires mechanisms for cross-region routing and optional KV replication or on-demand transfer.

- **Observability and operational tooling.** Production readiness requires detailed telemetry (per-sequence hit/miss rates, per-worker GPU memory/compute utilization, tail latency), alerting, and dashboards. For example, integrating Prometheus/Grafana, structured tracing, and cost monitoring will facilitate operations and capacity planning.

- **Tensor compression and dtype optimizations.** Reducing the network and memory footprint of KV tensors (e.g., `float16/bfloat16`, quantization, or lightweight compression) can lower costs and improve throughput without significantly harming model quality.

- **Model-specific and kernel-level optimizations.** Integrating the cache with model-specific attention kernels (vLLM-style paging, Triton/TensorRT fused operators) and GPU-side memory management can further reduce overhead and unlock higher throughput.

- **Security and production hardening.** Strengthening ingress authentication (IAP, API keys, OAuth), enabling mTLS or a service mesh for pod-to-pod encryption, and enforcing least-privilege IAM roles will be important for production deployments.

In summary, our work demonstrates that treating attention KV state as a distributed system resource — collocated with GPU compute and discoverable via a lightweight routing layer — is a practical and effective approach for scaling LLM inference in cloud environments. The proposed future directions provide a road map to improve cost-efficiency, robustness, and global scalability for latency-sensitive LLM services.

All source code, deployment scripts, and Kubernetes/Terraform configurations are available in an open-source GitHub repository[1] [NG25] to support reproducibility and future extensions.

[1]`https://github.com/ndminhvn/distributed-kv-cache`

Bibliography

[DFE$^+$22] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[KLL$^+$97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 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*, pages 654–663, 1997.

[KLZ$^+$23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[LV14] John Lamping and Eric Veach. A fast, minimal memory, consistent hash algorithm, 2014.

[MZM$^+$25] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow, 2025.

[NG25] Minh Nguyen and Zhenghui Gui. Distributed inference key-value cache in cloud setting. `https://github.com/ndminhvn/distributed-kv-cache`, 2025.

[YCL$^+$25] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025.