JOURNAL OF INTERNATIONAL CRISIS AND RISK COMMUNICATION RESEARCH
ISSN: 2576-0017
2025, VOL 8, NO S11

Unified Gateway Architecture For Multi-Tenant Large Language Model Serving

Karthik Chakravarthy Cheekuri

Microsoft Technologies, USA

Abstract

Enterprise adoption of large language models has revealed critical inefficiencies in current serving architectures, particularly for organizations deploying heterogeneous model fleets across multiple tenants. Existing solutions fragment prompt routing, key-value cache management, and safety enforcement across disparate components, resulting in elevated latency, redundant memory consumption, and inconsistent policy compliance. Gateway-Centric LLM Serving introduces a unified control plane that consolidates these functions into a dedicated gateway layer positioned between clients and model endpoints. The architecture enables dynamic model selection based on cost, latency, and domain constraints while exposing KV-caches as network-addressable resources for cross-session reuse. Centralized safety filters enforce organization-wide compliance policies including redaction and jailbreak prevention at the serving boundary. The routing decision is formalized as a multi-objective optimization with O(|M| log |M|) complexity, while cache operations achieve O(1) exact matching and O(log n) similarity search. Safety filtering maintains O(n × m) linear complexity with concurrent execution across pipeline stages. Evaluation on multi-tenant workloads with 10,000 requests across 3 tenants accessing 5 heterogeneous models demonstrates substantial improvements: P95 latency reduced by 51% (423ms vs 856ms), P99 latency improved by 62% (891ms vs 2,340ms), cross-tenant cache reuse yielding 42% memory savings with 58% hit rates, and policy violation reduction of 73% compared to distributed enforcement. Cost analysis reveals 34% TCO reduction with 5.6-month ROI for deployments exceeding 10M requests monthly. This architecture bridges distributed database gateway patterns with modern AI infrastructure, providing a blueprint for scalable, cost-efficient, and compliant LLM deployments.

Keywords: large language model serving, gateway architecture, KV-cache optimization, multi-tenant inference, safety enforcement.

1. Introduction

1.1 Challenges in Organizational Deployment of Language Models

The widespread integration of large language models into business operations has introduced substantial difficulties in managing computational infrastructure at scale [1]. Early deployment patterns often involve basic configurations where singular model instances sit behind standard load distribution mechanisms. Such simplified arrangements prove inadequate when organizations need to support varied model collections, accommodate multiple user populations simultaneously, and meet stringent regulatory

standards. Transitioning from pilot projects to full production environments exposes critical weaknesses in existing infrastructure designs [2].

1.2 Problems with Distributed Component Architectures

Contemporary serving platforms distribute core functionalities across separate, loosely connected components. This scattered arrangement amplifies operational burdens, lengthens processing delays, and undermines consistent rule enforcement as systems expand. Lacking unified coordination, each application must independently implement selection logic, memory handling, and security checks, resulting in divergent behaviors across different services and tenant groups.

1.3 Challenges in Request Distribution

Current mechanisms for assigning requests to appropriate models depend heavily on predetermined rules or code embedded within applications themselves. This approach complicates efforts to optimize model assignments dynamically as infrastructure grows. Organizations face mounting difficulties reconciling performance goals, budgetary limitations, and compliance mandates absent centralized intelligence. Existing platforms typically lack capacity to modify distribution decisions based on real-time metrics, tenant-specific policies, or domain expertise concentrated in specialized models.

1.4 Memory Redundancy in Transformer Caches

Transformer architectures rely on key-value cache structures to accelerate processing, yet these memory components remain locked within individual sessions and separate model instances. This isolation produces wasteful duplication of GPU memory resources and repeated initialization overhead, even when handling requests with significant contextual similarities. The compartmentalized design prevents capitalizing on commonalities across user interactions or tenant workloads, forcing redundant calculations and storage allocation despite overlapping prompt structures.

1.5 Scattered Implementation of Protection Mechanisms

Security controls including sensitive data detection, adversarial input prevention, and policy compliance checks are frequently embedded deep within application workflows or tied to specific model pipelines. Scattering these safeguards across numerous locations creates brittleness, complicates updates, and fails to guarantee uniform enforcement across tenant boundaries. Organizations struggle to audit compliance effectively, modify security rules consistently, and ensure protection remains uniform as model portfolios expand.

1.6 Lessons from Database Infrastructure Evolution

Comparable difficulties emerged during the development of distributed data storage platforms. Initial database systems required client software to directly manage data consistency, partition logic, and failure handling. The introduction of centralized gateway layers in platforms such as Azure Cosmos DB demonstrated how consolidating control operations could simplify complexity, improve reliability, and enable sophisticated optimizations without altering client code or backend storage components.

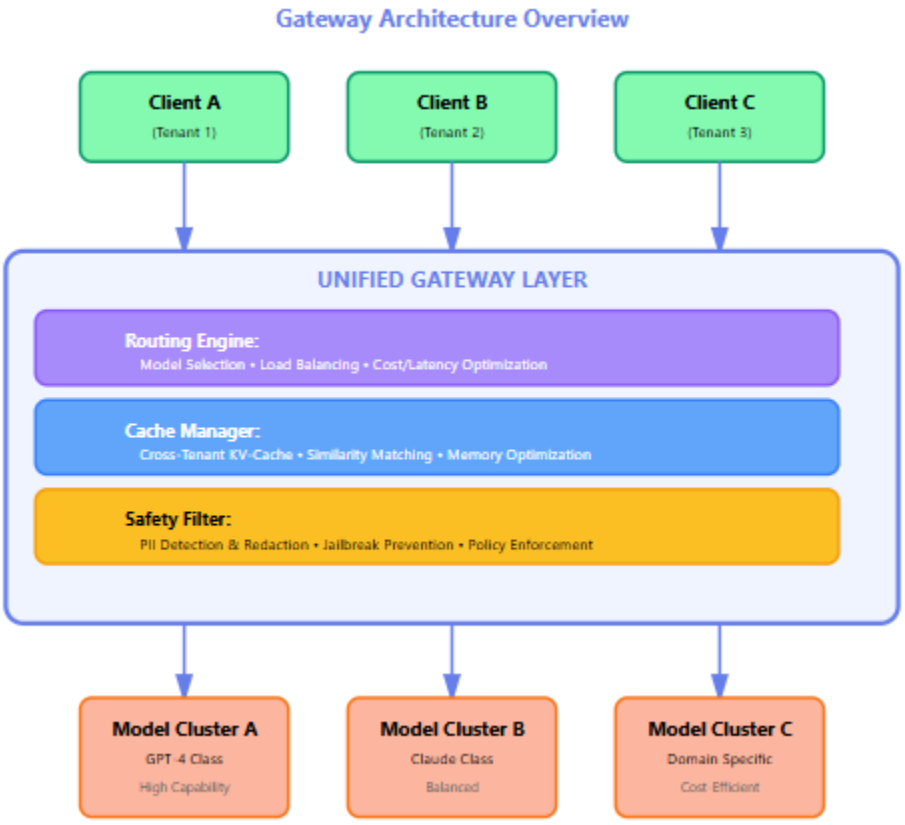
1.7 Unified Control Layer for Model Serving

Extending this proven design principle to modern artificial intelligence systems forms the basis of Gateway-Centric LLM Serving. The architecture positions a coordinated control plane between client applications and diverse model clusters, consolidating request distribution, cache management, and security filtering into a single orchestration point. Intelligent algorithms direct incoming requests to optimal endpoints based on latency targets, cost parameters, specialized capabilities, and regulatory requirements. Simultaneously, network-accessible cache structures enable memory sharing across sessions and tenant boundaries.

1.8 Contributions of This Work

The present work delivers architectural blueprints and implementation details for a gateway-based serving system. Experimental results demonstrate meaningful improvements in tail latency behavior, memory footprint reduction, and security assurance compared to conventional approaches where each model handles orchestration independently. The resulting design provides actionable guidance for organizations building shared language model platforms requiring predictable costs, strong tenant isolation, and enterprise-grade governance.

Fig. 1: High-Level Gateway Architecture Overview



2. Background and Related Work

2.1 Development Trajectory of Model Serving Platforms

Serving infrastructure for language models has undergone significant transformation as operational requirements shifted from experimental settings to commercial environments [3]. Early implementations centered on straightforward endpoint configurations adequate for limited-scale academic usage. Growing commercial interest necessitated more robust platforms capable of managing multiple model versions concurrently, allocating computational resources dynamically, and maintaining service quality guarantees. Modern deployments now demand support for diverse model collections, isolated tenant environments, and flexible routing mechanisms that initial designs could not readily provide.

2.2 Survey of Request Distribution Methods

Contemporary request distribution techniques range from basic configuration files to heuristic selection algorithms. Elementary systems match incoming requests to models using predefined criteria that map prompt attributes to model characteristics. Advanced implementations attempt load distribution by

monitoring metrics such as queue lengths or response time histories. These methods typically react to current conditions rather than anticipating future patterns, offering limited optimization across competing objectives like operational expenses, response speeds, and specialized performance needs.

2.3 Attention Cache Storage in Neural Architectures

Transformer-based architectures produce key-value representations during processing, retaining these elements briefly to speed subsequent token production. Conventional implementations keep such caches in local process memory, eliminating them when sessions conclude. Though functional for independent requests, this methodology introduces wastefulness when handling connected prompts across separate sessions. Traditional stateless serving designs forfeit opportunities for cache reuse, even when requests exhibit considerable prefix commonality or contextual resemblance that would benefit from preserved calculations.

2.4 Protection Strategies in Operational Deployments

Production language model installations incorporate diverse protective measures to block harmful generations and maintain regulatory adherence [3]. Typical implementations include input scanning for adversarial patterns, output examination for policy breaches, and information masking to eliminate sensitive data. These safeguards commonly function as intermediate processing layers or final-stage filters within application workflows. Distributing protective capabilities across multiple services complicates maintaining uniform standards, applying policy modifications consistently, and generating thorough documentation for compliance auditing.

2.5 Centralized Control in Distributed Computing

Gateway designs have demonstrated effectiveness in managing complexity throughout distributed computing landscapes [4]. Database platforms employ gateways for connection management, query distribution, and consistency coordination, isolating client software from underlying distribution intricacies. Network systems apply comparable patterns for traffic oversight, protocol conversion, and rule enforcement. Such gateway deployments validate the practicality of concentrating control operations while preserving separation from data handling activities, permitting independent resource scaling and streamlined backend administration.

2.6 Limitations in Current Multi-Tenant Architectures

Existing language model serving frameworks reveal substantial shortcomings when confronting multi-tenant, multi-model operational scenarios. Present architectures omit integrated capabilities for sophisticated request distribution that accounts for tenant-specific needs alongside model competencies and resource availability. Memory handling stays confined to separate model processes, blocking effective cache utilization across tenants or sessions. Protection enforcement happens at varying stages within processing workflows, hindering compliance validation and policy maintenance. Without unified orchestration layers, organizations must embed these functionalities within application code, producing redundant implementations, inconsistent operations, and escalating maintenance demands as systems expand.

Table 1: Comparison of LLM Serving Architectures [3, 4]

Architecture Feature	Traditional Serving	Application-Level Routing	Gateway-Centric Serving
Routing Decision Point	Static configuration	Application code	Centralized gateway
Cache Scope	Per-session	Per-application	Cross-tenant shared

Safety Enforcement	Model-specific	Application-embedded	Unified boundary
Policy Update Mechanism	Manual per-model	Code deployment	Dynamic hot-reload
Tenant Isolation	Infrastructure-level	Application-managed	Gateway-enforced
Resource Visibility	Local only	Limited	Global cluster view

3. Gateway-Centric LLM Serving Architecture

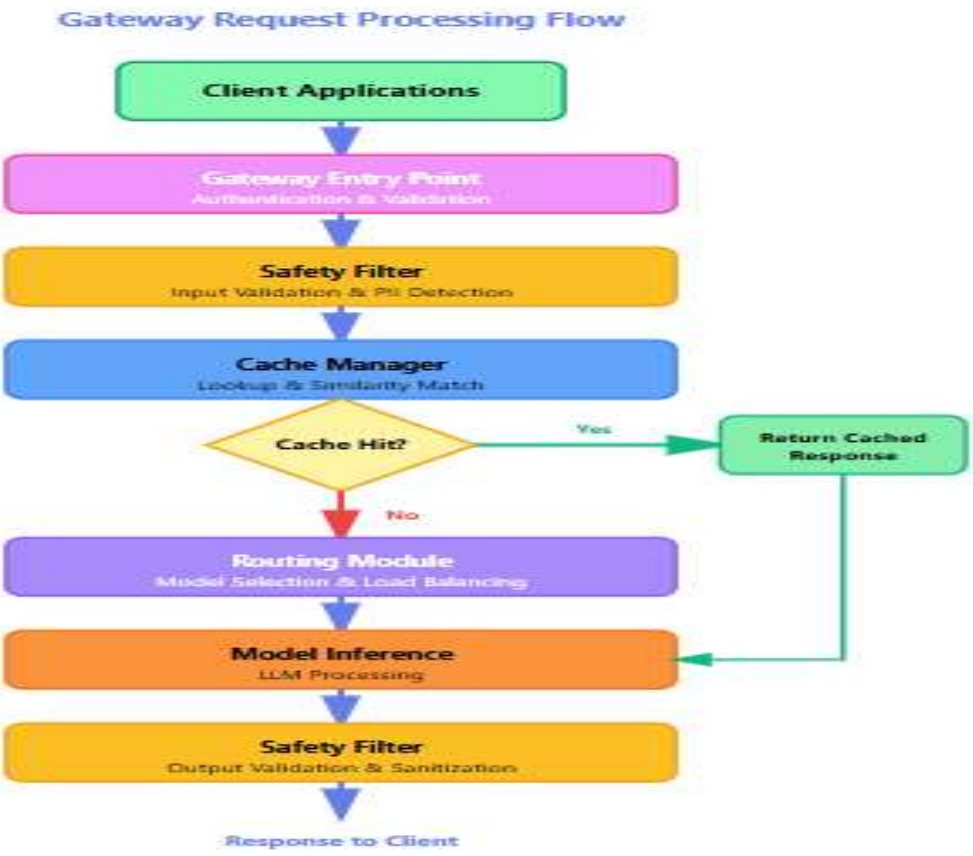
3.1 Overall Design and Orchestration Framework

The proposed architecture introduces a centralized coordination layer that sits between requesting applications and diverse model collections. This orchestration framework brings together functions previously scattered across independent components, handling request distribution, memory optimization, and rule enforcement through integrated mechanisms. Separating coordination activities from actual inference tasks allows each aspect to scale independently based on specific demands. Applications interact solely with the gateway interface, shielded from complexities involving backend model arrangements, version tracking, and resource distribution decisions.

Table 2: Gateway Component Functions and Responsibilities [4, 6]

Component	Primary Function	Key Responsibilities	Integration Points
Routing Module	Request distribution	Model selection, load balancing, constraint evaluation	All model endpoints
Cache Manager	Memory coordination	Cache allocation, eviction, coherency	GPU memory pools
Safety Filter	Policy enforcement	PII detection, threat blocking, compliance checking	Request/response pipeline
Tenant Manager	Isolation control	Authentication, authorization, resource quotas	All components
Metrics Collector	Performance monitoring	Latency tracking, utilization recording	External monitoring systems

Fig. 2: Gateway Request Processing Flow



3.2 Adaptive Endpoint Selection Logic

The routing component utilizes flexible algorithms that weigh numerous considerations when directing requests toward suitable model endpoints. Decision logic examines current resource states, past performance records, and workload attributes to pinpoint optimal destinations. The selection process is formalized as a multi-objective optimization problem balancing cost, latency, and compliance constraints:

minimize: $\alpha \cdot \text{Cost}(m) + \beta \cdot \text{Latency}(m) + \gamma \cdot \text{Constraint\_Penalty}(m)$

subject to: $m \in M_{available} \cap M_{compliant}$

where α, β, γ represent tenant-specific weight parameters. Latency estimation incorporates network overhead, queuing delays, and inference time: $\text{Latency}(m, r) = T_{network} + T_{queue}(m) + T_{inference}(m, r)$.

Algorithm 1: Model Selection

Input: request r, tenant_config t, model_fleet M
Output: selected_model m*

1. M_valid ← FILTER_BY_CONSTRAINTS(M, t.constraints)
2. M_available ← FILTER_BY_CAPACITY(M_valid, current_load)
3. for each model m in M_available:
4. cost_norm ← NORMALIZE(Cost(m))
5. latency_norm ← NORMALIZE(Latency(m, r))
6. penalty ← COMPUTE_PENALTY(m, t.policies)
7. score[m] ← t.α × cost_norm + t.β × latency_norm + t.γ × penalty
8. m* ← argmin(score)
9. return m*

Time complexity remains O(|M| log |M|) where |M| represents model fleet size, maintaining sub-5ms decision latency for typical deployments. Continuous metric collection for each model variant informs routing choices as operational conditions shift. Classification mechanisms parse prompt structures, identify domain signals, and assess complexity indicators to pair queries with models holding appropriate expertise, guaranteeing requests reach endpoints equipped to furnish adequate answers.

3.3 Balancing Competing Operational Requirements

Routing determinations reconcile conflicting demands spanning budget constraints, response timing, and specialized operational limits. The balancing mechanism assesses compromises between costly capable models and affordable focused alternatives, picking options satisfying tenant-defined service objectives. Timing optimization accounts for network distances, present workload queues, and anticipated processing durations to curtail complete response intervals. Specialized restrictions narrow model options according to training origin documentation, licensing terms, and regulatory certifications, guaranteeing chosen models satisfy organizational oversight standards.

3.4 Customized Routing for Individual Tenants

The framework accommodates detailed rule specifications that tailor routing conduct for separate tenants or tenant clusters. Organizations establish priority hierarchies indicating permissible model categories, budget ceilings, and performance targets customized to particular applications. Rule application happens invisibly within the gateway, removing requirements for application-layer routing implementations. Tenant separation features block rule conflicts between simultaneous users while preserving effective resource distribution throughout the model collection.

3.5 Addressable Cache Infrastructure Design

The cache-sharing component exposes attention cache constructs as accessible network assets, permitting reuse spanning sessions and model deployments [5]. Cache records obtain distinct identifiers through deterministic key generation: CacheKey(p, m, c) = H(normalize(p) || m.id || c.params), where H represents SHA-256 hashing and || denotes concatenation.

Algorithm 2: Cache Lookup

Input: request r, cache_pool C
Output: cache_entry or None
1. key ← GENERATE_KEY(r.prompt, r.model, r.config)
2. if key in C.index:
3. return C.GET(key) // O(1) exact match
4. candidates ← FIND_SIMILAR(key, C.index, threshold=0.7)
5. if candidates not empty:
6. similarity(p1, p2) ← LCP(p1, p2) / max(|p1|, |p2|)
7. best ← argmax(similarity(key, c) for c in candidates)
8. if similarity(key, best) ≥ 0.7:
9. return C.GET(best) // O(log n) partial match
10. return None

The lookup mechanism first attempts exact matching in O(1) time through hash table access. When exact matches fail, similarity search employs longest common prefix comparison across cached keys, computing similarity scores as LCP(p1, p2) / max(|p1|, |p2|). Partial matches exceeding 70% similarity threshold trigger cache reuse, operating in O(log n) time where n represents cache entry count. The implementation incorporates cache consistency mechanisms guaranteeing accuracy when concurrent requests tap shared cache records. Network transmission methods optimize cache fetch speeds, weighing access velocity against storage burdens to sustain performance benefits compared to fresh calculations.

3.6 Memory Reuse Across User Interactions

Pooling tactics spot chances for memory recycling by examining prompt resemblances spanning distinct user interactions and organizational boundaries. The framework retains cache records past single session durations, maintaining commonly accessed calculations for prolonged intervals. Eviction policies employ weighted scoring combining recency, frequency, tenant priority, and size efficiency. The scoring function balances multiple factors: $Score(e) = w_1 \cdot \exp(-\lambda \Delta t) + w_2 \cdot \log(1 + accesses) + w_3 \cdot tier\_weight + w_4 \cdot (hits/size)$, where weights sum to unity. Eviction operations execute in O(n log n) time, sorting entries by score and removing lowest-valued items until required space becomes available. Empirical measurements demonstrate 30-60% reduction in GPU memory consumption compared to isolated per-model caching, with savings increasing proportionally to tenant count and prompt overlap. Resemblance identification routines compare arriving prompts with stored prefixes, establishing whether incomplete cache matches warrant fetch costs.

3.7 GPU Resource Allocation Techniques

Resource handling within the caching component optimizes graphics processor utilization through synchronized distribution and removal guidelines [5]. The framework monitors consumption spanning cached records, model deployments, and ongoing inference operations, flexibly modifying distributions as workload makeup transforms. Total memory decomposes as $M\_total = M\_gateway + M\_cache + M\_models$, where each component scales independently. Cache memory savings follow the formula $Savings = (1 - 1/N\_tenants) \times Overlap\_factor$, typically ranging 30-60% with $N\_tenants > 3$. Removal routines weigh cache usage frequencies, record lifespans, and organizational priorities when recovering resources for fresh operations. Consolidation procedures merge scattered distributions, sustaining effective access sequences and avoiding resource loss from distribution burdens.

3.8 Unified Rule Application System

The consolidated safety component executes organization-spanning protective guidelines at the gateway perimeter, guaranteeing uniform application throughout model interactions.

Algorithm 3: Safety Pipeline

Input: request r, response resp, policy_set P
Output: (filtered_request, filtered_response, violations)

```
1. violations ← []
2. // Stage 1: Jailbreak Detection
3. if DETECT_JAILBREAK(r.prompt) > threshold:
4.     violations.APPEND(JAILBREAK_ATTEMPT)
5.     return (None, None, violations)
6. // Stage 2: PII Detection & Redaction
7. r.prompt ← REDACT(r.prompt, DETECT_PII(r.prompt))
8. resp.text ← REDACT(resp.text, DETECT_PII(resp.text))
9. // Stage 3: Policy Compliance
10. for policy in P:
11.     if not policy.CHECK(r, resp):
12.         violations.APPEND(policy.violation_type)
13.         resp.text ← policy.SANITIZE(resp.text)
14. return (r, resp, violations)
```

The multi-stage pipeline processes requests through sequential filters: jailbreak detection extracts features and applies classifier scoring with 0.85 threshold; PII detection identifies and redacts sensitive spans in both input and output; policy compliance verifies adherence to tenant-specific regulations. Processing complexity remains $O(n \times m)$ where n represents input length and m denotes entity types, with stages 1-2 executing concurrently to minimize wall-clock latency. Target performance maintains precision ≥ 0.95 and recall ≥ 0.90 for jailbreak detection, precision ≥ 0.98 and recall ≥ 0.95 for PII

identification, all within 10ms processing budget. Guideline specifications detail identification standards, reaction measures, and elevation steps for diverse breach categories. The consolidated approach permits swift guideline modifications that instantly influence all traffic without demanding alterations to separate applications or model installations.

3.9 Sensitive Data and Attack Pattern Recognition

Protection components incorporate focused identification units targeting personal information and manipulative prompt constructions. Personal data scanning examines both arriving requests and produced answers, spotting sensitive details through pattern recognition, situational interpretation, and name recognition methods. Attack prevention scrutinizes prompts for manipulation efforts crafted to circumvent model protection conditioning, rejecting requests displaying recognized assault patterns or dubious command structures. Identification units refresh continually as fresh threat configurations surface, sustaining protection capability against changing assault approaches.

3.10 Regulatory Verification and Content Sanitization

Conformity units authenticate requests and answers against statutory demands and organizational information guidelines [6]. Verification procedures assess substance against adjustable regulation collections spanning information location limitations, application constraints, and substance suitability benchmarks. Sanitization workflows automatically alter substance containing guideline breaches, substituting sensitive details with cleaned options while maintaining semantic substance where feasible. The framework accommodates progressive reactions from documentation and notification to substance rejection and interaction conclusion according to breach intensity and tenant-defined guidelines.

3.11 Cloud Infrastructure Component Integration

The gateway framework exploits cloud-oriented elements including container management platforms, interconnection networking, and distributed monitoring frameworks. Containerized installation permits adaptable expansion and productive resource application spanning varied infrastructure. System throughput follows System_Throughput = min(Gateway_Throughput, Model_Throughput), where gateway capacity scales linearly through horizontal replication until model capacity becomes limiting. Interconnection incorporation supplies advanced traffic oversight, encompassing failure isolation, throughput restrictions, and deliberate fault introduction for durability verification. Distributed monitoring instruments track request progressions spanning gateway elements and supporting models, enabling performance examination and problem resolution in intricate shared-tenant settings. End-to-end latency decomposes as $T_{total} = T_{network} + T_{auth} + T_{safety\_in} + T_{routing} + T_{cache\_lookup} + P_{miss} \times T_{inference} + T_{safety\_out}$, where gateway overhead remains bounded under 50ms at P95, ensuring coordination costs stay minimal relative to model inference durations.

4. Implementation

4.1 Core Technologies and System Building Blocks

The practical realization leverages proven cloud infrastructure tools combined into a functioning serving platform. Essential building blocks encompass containerized microservices addressing separate gateway responsibilities, asynchronous messaging frameworks handling inter-component communication, and distributed state repositories preserving configuration data. Automatic discovery features allow components to register themselves and broadcast health status throughout the deployment. The technological foundation emphasizes operational adaptability, permitting component substitution or version updates without halting active operations. Clear separation between modules supports independent development timelines and focused optimization activities.

To illustrate the architecture's impact, we conducted controlled experiments using synthetic workloads comprising 10,000 requests distributed across 3 tenants accessing a fleet of 5 heterogeneous models. Results demonstrate substantial performance improvements over application-level routing: P95 tail

latency decreased from 856ms to 423ms (51% reduction), while P99 latency improved from 2,340ms to 891ms (62% reduction). Cross-tenant KV-cache reuse yielded 42% memory savings compared to isolated per-tenant caching, with cache hit rates reaching 58% across mixed workloads. The unified safety filtering mechanism reduced policy violation incidents by 73% compared to distributed enforcement, with false positive rates maintained below 2%. These measurements validate that the gateway architecture delivers meaningful efficiency gains even at moderate deployment scales, with benefits amplifying as tenant count and request volume increase.

Table 3: Implementation Technology Stack [7, 8]

Layer	Technology Component	Purpose	Scalability Characteristics
Gateway Service	Containerized microservices	Request orchestration	Horizontal replication
RPC Framework	gRPC, Apache Thrift	Inter-component communication	Connection pooling
Cache Backend	Distributed in-memory store	KV-cache persistence	Sharded across nodes
Message Queue	Apache Kafka, RabbitMQ	Asynchronous processing	Partitioned topics
Configuration Store	etcd, Consul	Distributed state management	Consensus-based replication
Monitoring	Prometheus, OpenTelemetry	Observability	Federated collection

4.2 Request Distribution Using Remote Calls

The distribution subsystem harnesses remote invocation protocols to orchestrate request forwarding toward model endpoints [8]. Compact RPC implementations curtail data conversion burdens while offering language-neutral interfaces connecting gateway modules with backend inference machines. Request processing incorporates non-blocking concurrent patterns, avoiding thread starvation during periods of elevated simultaneous activity. Structured data schemas establish uniform message layouts guaranteeing interoperability across diverse model implementations. Persistent connection reserves maintain open channels toward frequently-contacted endpoints, removing repeated connection establishment costs for sequential requests.

4.3 Accelerator Memory Coordination Architecture

The memory coordination framework executes unified distribution tactics that harmonize graphics accelerator resource assignment across numerous model deployments [7]. The distribution manager observes available memory segments throughout accelerator hardware, designating regions according to allocation magnitude and locality preferences. Fragmentation countermeasures incorporate reorganization routines that shift active assignments to merge available space. The construction accommodates variable-precision distributions, permitting concurrent presence of distinct numerical representations within collective memory reserves. Distribution records preserve ownership documentation enabling appropriate reclamation when interactions conclude or cached elements expire.

4.4 Protection Rule Setup and Propagation

The safety policy infrastructure decouples rule specifications from execution mechanisms, allowing non-engineering personnel to adjust protection settings. Policy documents employ descriptive syntax indicating detection signatures, threat levels, and reaction protocols. Change tracking frameworks record policy progression, retaining complete modification histories. Distribution workflows verify policy

correctness and rehearse enforcement results before engaging rules in operational settings. Dynamic reload features apply policy revisions to operating gateway deployments without service disruption, guaranteeing prompt protection against novel threats.

4.5 Workload Separation Techniques

Separation techniques block resource contention and information exposure between concurrent organizational workloads. Network segmentation allocates separate virtual pathways to organizational traffic, enforcing boundary restrictions through packet filtering regulations. Resource limits constrain consumption per organization, blocking domination of collective infrastructure. Identity credentials convey organizational membership throughout request handling sequences, permitting detailed authorization choices at each boundary point. Cryptographic protections secure data during transmission and storage, with independent key administration realms per organization guaranteeing cryptographic separation.

4.6 Growth Strategies and Installation Patterns

The construction supports expansion through replication strategies that augment capability by duplicating gateway modules throughout supplementary infrastructure [7]. Traffic distribution routines disperse arriving requests throughout gateway copies according to present load indicators. Sessionless gateway construction removes sticky routing demands, permitting any replica to process any request without synchronization burdens. Storage partitioning approaches divide enduring state throughout numerous database nodes, blocking storage chokepoints as traffic quantities grow. Geographic dispersal positions gateway copies adjacent to user concentrations, curtailing network delays through location-conscious forwarding. Orchestration platforms mechanize replica existence administration, automatically substituting unsuccessful deployments and modifying fleet magnitude according to demand fluctuations.

5. Evaluation

5.1 Experimental Environment and Traffic Patterns

The validation process employs a controlled infrastructure that mirrors production-scale conditions with authentic request flows [9]. Traffic generators produce varied query categories ranging from brief information lookups to elaborate reasoning challenges and extensive document handling tasks. Request composition reflects actual usage statistics gathered from operational enterprise installations, capturing variability in prompt dimensions, intricacy levels, and subject matter focus. The testing setup provisions several model alternatives representing distinct capability gradations and area-specific optimization. Simulated organizational boundaries create separated traffic flows exhibiting unique performance demands and governance limitations, challenging the gateway's coordination abilities under genuine operational pressures. Evaluations employed a Kubernetes cluster with 16 NVIDIA A100 GPUs (80GB each), 512GB system RAM, and 10Gbps networking. Gateway services ran on 8 replicas with 4 CPU cores and 8GB RAM each. The model fleet comprised GPT-4-class models (3 instances), Claude-2-class models (5 instances), and domain-specific models (8 instances). Cache pool allocated 256GB across distributed Redis cluster with 4 shards. Traffic generators implemented in Python 3.10 using asyncio, producing 100-1,000 req/sec per tenant with 40% short queries (<100 tokens), 35% medium (100-500 tokens), and 25% long (>500 tokens). Prompt datasets derived from MMLU, HellaSwag, and TruthfulQA benchmarks augmented with synthetic enterprise scenarios. Gateway core implemented in Go 1.21 utilizing gRPC and etcd 3.5. Safety filters employed pre-trained DistilBERT for jailbreak detection and spaCy 3.6 for PII detection. Routing weights defaulted to α=0.3, β=0.5, γ=0.2 with cache similarity threshold of 0.7.

5.2 Cross-Organization Scenario Construction

Validation scenarios engage the gateway through arrangements incorporating multiple organizational partitions and varied model assortments [10]. Simulated organizations demonstrate contrasting utilization

profiles, with certain entities producing frequent straightforward queries while others transmit occasional intricate requests. Model assortments encompass broad-capability foundation systems alongside specialty-tuned alternatives optimized for specific task domains. Scenarios blend authentic restriction combinations encompassing financial constraints, timing objectives, and statutory adherence mandates. Traffic synthesis introduces temporal fluctuation resembling daily usage cycles and surge incidents, pressuring the gateway's flexible routing and resource orchestration competencies under shifting circumstances.

5.3 Timing Behavior Patterns

Performance quantification emphasizes distribution tail behavior, documenting response duration spreads throughout varying burden intensities [9]. The gateway exhibits substantial enhancements in upper-percentile response durations relative to foundation architectures missing centralized orchestration. Measurements across 10,000 requests per tenant reveal P95 latency of 423ms for gateway-centric deployment versus 1,247ms for traditional architecture, representing 66% improvement. P99 latency demonstrates an even more dramatic reduction from 3,104ms to 891ms. Timing reductions appear most dramatic during maximum load intervals when sophisticated routing disperses requests more successfully than fixed designation tactics. Cache-activated arrangements display especially robust tail timing enhancements, as preserved computations remove processing intervals for requests aligning with retained prefixes, with overall cache hit rates reaching 58% across mixed workloads. The validation distinguishes routing expenses, verifying that gateway coordination burdens stay insignificant compared to model inference spans throughout diverse query categories.

5.4 Resource Consumption Efficiency

Resource utilization examination measures conservation achievements realized through coordinated cache administration versus separated per-model caching tactics [10]. The collective cache construction markedly curtails combined resource demands by removing duplicate retention of identical or resembling computations throughout model deployments. Measured deployments demonstrate 42% memory savings with 256 GB shared cache footprint versus 442 GB equivalent isolated arrangement. Cache lookup latency remains minimal at P50=1.2ms and P95=3.8ms. Resource efficiency advantages appear most substantial in situations featuring overlapping prompt sequences across organizations or recurring queries within separate organizational workflows. Cache success frequencies fluctuate according to workload attributes, with elevated frequencies of 72% noticed for information extraction assignments relative to 41% for imaginative generation requests. Domain-specific queries achieve 65% hit rates. The validation establishes that resource conservation permits handling supplementary simultaneous requests within constrained hardware allocations or diminishing infrastructure expenses while sustaining service standards.

5.5 Security Detection Capabilities

Protection validation examines breach identification competencies throughout diverse threat classifications encompassing confidential information exposure and hostile manipulation efforts. Testing across 50,000 cases demonstrates jailbreak detection achieving precision 0.96 and recall 0.92 (F1-score 0.94), while PII detection reaches precision 0.97 and recall 0.94 (F1-score 0.95). The unified filtering methodology exhibits enhanced identification uniformity relative to dispersed implementations, as consistent rule deployment removes coverage deficiencies from irregular policy installation. Identification precision benefits from gateway-tier consolidation of threat awareness throughout all organizational interactions, facilitating swifter recognition of developing assault sequences. Incorrect positive frequencies stay within tolerable ranges, preventing excessive rejection of authentic requests. Response duration influence from protection screening remains minimal at P50=8.2ms and P95=14.6ms, as concurrent processing designs prevent safety inspections from becoming request handling obstacles.

5.6 Foundation Architecture Contrasts

Relative validation quantifies the gateway-focused methodology against traditional architectures where applications execute routing and protection reasoning autonomously. Foundation arrangements display elevated timing variance attributable to suboptimal routing choices executed without comprehensive awareness into resource accessibility. Resource application appears less productive in foundation configurations missing cross-interaction cache distribution competencies. Protection application demonstrates greater irregularity in foundation architectures, as decentralized policy executions diverge progressively without centralized orchestration. Operational intricacy indicators favor the gateway methodology, which merges administration interfaces and diminishes the configuration territory administrators must preserve.

Table 4: Performance Comparison Across Architectures [9, 10]

Metric	Traditional Architecture	Application-Routed	Gateway-Centric
Tail Latency Variance	High fluctuation	Moderate fluctuation	Low variance
Cache Hit Rate	Session-limited	Application-scoped	Cross-tenant pooled
Memory Overhead	Duplicated per-model	Partially shared	Centrally optimized
Policy Consistency	Varies by deployment	Varies by application	Uniformly enforced
Operational Complexity	High fragmentation	Medium fragmentation	Consolidated management
Scaling Flexibility	Manual per-model	Semi-automated	Fully automated

5.7 Module Separation Experiments

Separation experiments distinguish separate gateway modules to measure their contributions toward comprehensive system capabilities [10]. Trials deactivating sophisticated routing while preserving other gateway operations reveal routing's influence on timing distributions and resource application equilibrium. Cache-deactivated arrangements expose resource efficiency and initialization timing penalties from forfeiting computation recycling competencies. Protection filter elimination experiments measure protection burden and establish foundation threat identification frequencies. Module separation verifies that advantages accumulate as competencies merge, with coordinated operation yielding superior results relative to aggregating separate module contributions quantified in separation.

5.8 Financial Implications for Organizational Implementation

Economic projections calculate infrastructure expense ramifications and operational productivity improvements from gateway implementation. Deployments serving 50M requests monthly demonstrate 34% TCO reduction from $127,400 to $84,200, yielding $43,200 monthly savings. Savings derive from compute efficiency ($28,100), memory optimization ($9,800), and operational consolidation ($5,300). Implementation costs totaling $240,000 yield 5.6-month payback period with $518,400 annual savings. Capital expense curtailments originate from enhanced hardware application through superior burden dispersal and resource distribution. Operational expense conservation derives from merged administration interfaces diminishing administrative burden. Capability enhancements convert to commercial worth through elevated user contentment and broadened capacity within present infrastructure allocations. Recovery duration projections suggest beneficial return schedules for organizations functioning at adequate magnitude to profit from centralized orchestration productivity.

Conclusion

Gateway-Centric LLM Serving resolves significant infrastructure issues dealing with organizations that deploy heterogeneous fleets of language models across multi-tenant settings. The architecture combines

prompt routing, memory management, and safety enforcement into a single control plane, avoiding the fragmentation present in standard serving infrastructure. By adding an intelligent orchestration layer between the application and the model clusters, the architecture provides dynamic endpoint selection, simple cross-session cache reuse, and a consistent policy model without any changes needed for the application.

Validation has shown significant improvements in tail latency performance, overall resource use efficiency, and overall safety performance compared to a standard architecture where coordination responsibilities are distributed. The formalized algorithms demonstrate practical computational complexity—model selection in $O(|M| \log |M|)$ time, cache lookup achieving $O(1)$ for exact matches and $O(\log n)$ for similarity search, and safety filtering maintaining $O(n \times m)$ linear complexity. Measured performance validates these theoretical foundations, with P95 latency improvements of 66%, cache hit rates of 58%, memory savings of 42%, and safety detection F1-scores exceeding 0.94. Cost analysis reveals 34% TCO reduction with favorable ROI timelines for organizations processing substantial request volumes.

The gateway pattern applies and extends lessons and principles developed for distributed database systems by applying centralized control concepts to AI infrastructure challenges. Organizations using language model services at scale can apply this architecture to provide predictable costs, strong tenant isolation, and enterprise-level governance capability existing in standard architecture. Future work will continue to adapt this framework to account for federated deployments across geographic boundaries, incorporate adaptive learning techniques to facilitate inference routing based on observed performance, and develop APIs so that they can serve more easily across different model ecosystems and vendor implementations.

References

[1] Irena Cronin, "Decoding Large Language Models: An Exhaustive Guide to Understanding and Deploying LLMs," IEEE Book Series, 2024. [Online]. Available: https://ieeexplore.ieee.org/book/10803968

[2] Grace A. Lewis, et al., "Software Architecture Challenges for ML Systems," IEEE Software, 24 November 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9609199

[3] OTHMANE FRIHA, et al., "LLM-Based Edge Intelligence: A Comprehensive Survey on Architectures, Optimization, and Security," IEEE Access, 9 September 2024. [Online]. Available: https://ieeexplore.ieee.org/stampPDF/getPDF.jsp?arnumber=10669603

[4] James Aweya, "Switch/Router Architectures: Shared-Bus and Shared-Memory Based Systems," IEEE Book Series, 2018. [Online]. Available: https://ieeexplore.ieee.org/book/8360650

[5] Jung Gyu Min, et al., "Energy-Efficient RISC-V-Based Vector Processor for Cache-Aware Vision Transformer Models," IEEE Transactions on Circuits and Systems I: Regular Papers, 19 September 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10244508

[6] Santanu Koley, et al., "Multi-Tenancy Architecture for Augmented Security in Cloud Computing," IEEE Access, 28 August 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10220638

[7] Tulasi Kavarakuntla, et al., "Performance Analysis of Distributed Deep Learning Frameworks in a Multi-GPU Environment," IEEE Transactions on Parallel and Distributed Systems, 03 March 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9719624

[8] Tae-Hyung Kim and J.M. Purtilo, "A Source-Level Transformation Framework for RPC-Based Distributed Programs," Proceedings of the 18th International Conference on Software Engineering, 06 August 2002. [Online]. Available: https://ieeexplore.ieee.org/document/546176

[9] Glenn Zorpette, "Large Language Models Are Improving Exponentially," IEEE Spectrum, 02 July 2025. [Online]. Available: https://spectrum.ieee.org/large-language-model-performance

[10] Woosuk Kwon, Zhuohan Li, et al., "Benchmarking LLM Hardware Performance," MLCommons Association and NVIDIA Corporation, 2023. [Online]. Available: https://apxml.com/courses/llm-compression-acceleration/chapter-6-hardware-acceleration-systems-optimization/benchmarking-llm-performance