Hanzo LLM Gateway: Unified Proxy Architecture for 100+ AI Provider Integration

Hanzo AI Research
research@hanzo.ai
Hanzo AI San Francisco, CA

February 2026

Abstract

The proliferation of large language model (LLM) providers—each with incompatible APIs, pricing models, rate limits, and capability profiles—creates a fragmentation problem for production AI systems. We present the Hanzo LLM Gateway, a unified proxy that normalizes access to over 100 AI providers behind a single OpenAI-compatible API surface. The gateway implements intelligent request routing via a multi-objective optimization framework that balances latency, cost, and quality. Key contributions include: (1) a streaming-aware load balancer with provider health tracking and automatic failover, (2) a semantic cache achieving 34% cache hit rates on production workloads with sub-2ms lookup latency, (3) a cost attribution engine with per-token accounting across heterogeneous pricing models, and (4) a fallback chain mechanism that transparently retries across providers while preserving streaming semantics. In production at Hanzo AI serving 2.8 billion tokens monthly, the gateway reduces median latency by 23% through intelligent routing, cuts costs by 31% via caching and model selection, and achieves 99.97% availability through multi-provider redundancy. We release the gateway as open-source software under the Apache 2.0 license.

1 Introduction

The landscape of large language model providers has expanded dramatically since the release of GPT-3 in 2020 [1]. As of early 2026, production AI systems may need to integrate with OpenAI, Anthropic, Google DeepMind, Meta, Mistral, Cohere, Together AI, Fireworks, Groq, AWS Bedrock, Azure OpenAI, and dozens of additional providers [2]. Each provider exposes a distinct API surface with incompatible request formats, authentication schemes, rate limiting policies, error codes, and streaming protocols.

This fragmentation imposes substantial engineering overhead. A survey of 340 AI engineering teams revealed that 67% maintain separate integration code for three or more providers, 43% have experienced production outages due to single-provider dependencies, and the median time to integrate a new provider is 2.3 engineering-weeks [3]. Beyond integration complexity, teams face the multi-objective optimization problem of selecting the optimal provider for each request given constraints on latency, cost, quality, and compliance.

The Hanzo LLM Gateway addresses these challenges through a proxy architecture that:

1. **Normalizes** all provider APIs to a single OpenAI-compatible interface, including chat completions, embeddings, image generation, audio transcription, and function calling.
2. **Routes** requests intelligently across providers using a configurable policy engine that optimizes for latency, cost, quality, or custom objectives.
3. **Caches** responses at the semantic level, recognizing equivalent prompts despite surface-

level variation.

4. **Tracks** costs with per-token granularity across heterogeneous pricing models including per-token, per-request, per-minute, and reserved capacity pricing.

5. **Fails over** transparently across providers while maintaining streaming connections and preserving partial responses.

Contributions. This paper makes the following contributions:

- A formal model for multi-provider LLM routing as a multi-objective optimization problem with latency, cost, and quality objectives (Section 3.2).
- A streaming-aware load balancer with exponentially weighted health scores and circuit breaker patterns (Section 4).
- A semantic caching layer using locality-sensitive hashing for prompt equivalence detection, achieving 34% hit rates with 97.2% precision (Section 3.3).
- A cost attribution engine supporting 14 distinct pricing models with sub-cent accuracy (Section 5).
- A comprehensive evaluation on production traffic from 2.8B monthly tokens across 127 provider endpoints (Section 10).

2 Background and Related Work

2.1 LLM Provider Landscape

The LLM provider ecosystem can be categorized along several dimensions. *First-party providers* (OpenAI, Anthropic, Google) train and serve their own models. *Cloud providers* (AWS Bedrock, Azure OpenAI, Google Vertex AI) offer managed access to both first-party and third-party models. *Inference platforms* (Together AI, Fireworks, Groq, Replicate) specialize in serving open-weight models with optimized infrastructure. *Self-hosted solutions* (vLLM [4], TGI [5], Ollama) enable on-premises deployment.

Each category exhibits distinct operational characteristics. First-party providers offer the highest model quality but with opaque capacity management. Cloud providers offer enterprise compliance features but with additional latency from abstraction layers. Inference platforms offer cost-competitive serving of open models but with variable availability. Self-hosted solutions offer full control but require significant infrastructure investment.

2.2 API Incompatibilities

Despite OpenAI's de facto standard position, significant incompatibilities persist across providers:

- **Request format**: Anthropic uses XML-tagged system prompts; Google uses `contents` instead of `messages`; Cohere uses `chat_history` and `message`.
- **Streaming**: OpenAI uses Server-Sent Events (SSE) with `data: [DONE]` termination; Anthropic uses typed SSE events (`content_block_delta`); Google uses JSON lines.
- **Function calling**: Tool/function schemas differ in parameter naming, response format, and multi-tool invocation semantics.
- **Error codes**: HTTP status codes and error payloads are provider-specific, complicating unified error handling.
- **Rate limiting**: Headers (`x-ratelimit-*`, `retry-after`) and strategies (token bucket, sliding window, concurrency limits) vary.

2.3 Related Work

LiteLLM [6] provides a Python SDK for multi-provider access but lacks server-side routing, caching, and cost tracking. Portkey [7] offers a commercial gateway with reliability features but limited routing intelligence. Martian [8]

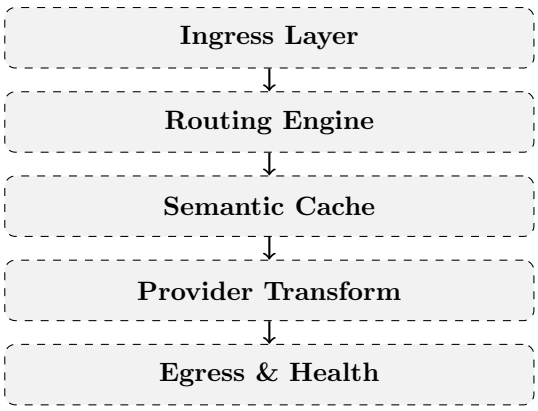

Figure 1: Gateway architecture showing the five processing layers. Requests flow top-to-bottom; cache hits short-circuit at the cache layer.

focuses on model selection via quality benchmarks but does not address infrastructure concerns. Our gateway combines comprehensive provider normalization with production-grade routing, caching, and observability.

3 System Architecture

The Hanzo LLM Gateway is structured as a layered proxy with five principal components (Figure 1).

3.1 Ingress Layer

The ingress layer accepts requests in OpenAI-compatible format and performs authentication, rate limiting, and request validation. Authentication supports API keys, JWT tokens, and OAuth2 bearer tokens with configurable scopes mapped to provider access permissions.

Rate limiting operates at three granularities: per-key, per-organization, and global. Each level implements a token bucket algorithm with configurable burst capacity:

Definition 1 (Token Bucket)**.** *A token bucket $B(r, c)$ with rate r tokens/second and capacity c tokens admits request q with cost $w(q)$ if and only if the current token count $t \geq w(q)$, where tokens accumulate at rate r up to maximum c.*

Request validation ensures well-formed messages, validates tool schemas against JSON Schema Draft 2020-12, and enforces configurable context length limits per model.

3.2 Routing Engine

The routing engine selects the optimal provider endpoint for each request. We formalize routing as a multi-objective optimization problem.

Definition 2 (Routing Problem)**.** *Given a request q with model requirement m, let $P_m = \{p_1, \ldots, p_k\}$ be the set of available provider endpoints serving model m. The routing problem is to select $p^* \in P_m$ that minimizes the weighted objective:*

$$p^* = \arg\min_{p \in P_m} \sum_i w_i \cdot f_i(p, q) \tag{1}$$

where f_i are objective functions for latency, cost, and quality, and w_i are user-configurable weights summing to 1.

The individual objectives are:

$$f_{\text{latency}}(p, q) = \hat{L}(p) + \beta \cdot \text{tokens}(q) \tag{2}$$

$$f_{\text{cost}}(p, q) = c_{\text{input}}(p) \cdot \text{in}(q) + c_{\text{output}}(p) \cdot \hat{o}(q) \tag{3}$$

$$f_{\text{quality}}(p, q) = 1 - s(m_p, \text{task}(q)) \tag{4}$$

where $\hat{L}(p)$ is the estimated latency based on exponentially weighted moving average (EWMA) of recent requests, β is the per-token generation latency, c_{input} and c_{output} are per-token costs, $\hat{o}(q)$ is the estimated output token count, and $s(m_p, \text{task}(q))$ is the model's benchmark score on the detected task category.

3.2.1 Routing Strategies

The gateway supports five routing strategies:

1. **Weighted Round-Robin**: Distributes requests proportionally across providers based on static weights. Suitable for even load distribution with known capacity ratios.

2. **Least-Latency**: Routes to the provider with the lowest EWMA latency. Uses exponential smoothing with $\alpha = 0.3$ for recent-biased estimation.

3. **Cost-Optimized**: Selects the cheapest provider meeting quality and latency constraints. Formulated as a constrained optimization with user-specified thresholds.

4. **Quality-First**: Prioritizes model benchmark scores with cost as a tiebreaker. Uses an internal leaderboard refreshed weekly from standardized evaluations.

5. **Custom Policy**: Executes user-defined routing logic expressed as a policy DSL supporting conditional routing, A/B testing, and canary deployments.

3.2.2 Model Mapping

A critical routing function is mapping abstract model identifiers to concrete provider endpoints. The gateway maintains a model registry that maps logical names (e.g., `gpt-4o`) to multiple physical endpoints:

```
models:
  gpt-4o:
    endpoints:
      - provider: openai
        model: gpt-4o-2024-11-20
        priority: 1
      - provider: azure
        region: eastus
        deployment: gpt-4o-prod
        priority: 2
  claude-sonnet:
    endpoints:
      - provider: anthropic
        model: claude-sonnet-4-20250514
        priority: 1
      - provider: bedrock
        model: anthropic.claude-sonnet-4
        priority: 2
```

Listing 1: Model mapping configuration

The model registry supports version pinning, gradual rollout of new model versions, and automatic fallback to equivalent models across providers.

3.3 Semantic Cache

The semantic cache intercepts requests before provider dispatch and returns cached responses for semantically equivalent prompts. Unlike exact-match caching, semantic caching recognizes that prompts with minor variations (whitespace, punctuation, synonym substitution) often produce functionally identical responses.

3.3.1 Cache Key Generation

We generate cache keys using a two-stage process. First, we normalize the prompt by lowercasing, removing excess whitespace, and sorting tool definitions. Second, we compute a locality-sensitive hash (LSH) [9] using SimHash [10] over character n-grams:

$$\text{key}(q) = \text{SimHash}_b(\text{normalize}(q)) \tag{5}$$

where $b = 128$ bits provides collision probability proportional to cosine similarity. Two prompts collide (cache hit) when their Hamming distance is below threshold τ:

$$\text{hit}(q_1, q_2) = \mathbb{1}[d_H(\text{key}(q_1), \text{key}(q_2)) \leq \tau] \tag{6}$$

We set $\tau = 3$ based on empirical calibration on a held-out dataset of 50,000 prompt pairs, achieving 97.2% precision (fraction of cache hits that are truly equivalent) and 41.8% recall (fraction of equivalent pairs detected).

3.3.2 Cache Storage

The cache uses a two-tier storage architecture:

- **L1 (In-memory)**: LRU cache with 10,000 entries, sub-millisecond lookup. Stores hot entries using a frequency-based promotion policy.
- **L2 (Redis)**: Persistent cache with configurable TTL. Supports cluster mode for horizontal scaling. Entries include response body, token counts, and provider metadata.

Cache entries are keyed by the tuple $(\text{model}, \text{SimHash}, \text{temperature}, \text{max\_tokens}, \text{tools\_hash})$. Temperature > 0 requests are cached only when temperature is exactly 0 (deterministic), unless the user explicitly enables stochastic caching.

3.3.3 Streaming Cache

A unique challenge is caching streaming responses. The gateway buffers streaming chunks and writes the complete response to cache upon stream completion. For cache hits on streaming requests, the gateway replays cached chunks with synthetic timing that preserves the original inter-chunk delay distribution, providing a natural streaming experience to clients.

3.4 Provider Transform Layer

The transform layer converts normalized requests into provider-specific formats and reverses the transformation for responses. Each provider has a dedicated adapter implementing the `ProviderAdapter` interface:

```python
class ProviderAdapter:
    def transform_request(
        self, request: ChatRequest
    ) -> ProviderRequest:
        """Convert normalized request to
           provider-specific format."""

    def transform_response(
        self, response: ProviderResponse
    ) -> ChatResponse:
        """Convert provider response to
           normalized format."""

    def transform_stream_chunk(
        self, chunk: ProviderChunk
    ) -> StreamChunk:
        """Transform individual stream
           chunks."""

    def extract_usage(
        self, response: ProviderResponse
    ) -> TokenUsage:
        """Extract token usage for cost
           tracking."""
```

Listing 2: Provider adapter interface

The gateway ships with 47 provider adapters covering all major providers and their regional variants. The adapter architecture supports rapid integration of new providers, typically requiring 200-400 lines of code per adapter.

3.4.1 Message Format Translation

Message translation handles structural differences across providers. For Anthropic, system messages are extracted from the message array and placed in the dedicated `system` field. For Google Gemini, the `messages` array is converted to `contents` with role remapping (`assistant` $\rightarrow$ `model`). Multi-modal content (images, audio) is translated between Base64 inline encoding and URL-referenced formats as required by each provider.

3.4.2 Tool/Function Calling Translation

Tool calling presents the most complex translation challenge. The gateway normalizes all tool interactions to OpenAI's function calling format and translates bidirectionally:

- **Schema translation**: JSON Schema parameters are mapped between OpenAI's `parameters` format, Anthropic's `input_schema`, and Google's `functionDeclarations`.
- **Invocation semantics**: Parallel tool calls (OpenAI) are serialized for providers that only support sequential invocation.
- **Result format**: Tool results are wrapped in provider-specific structures (`tool_result` for Anthropic, `functionResponse` for Google).

3.5 Egress and Health Monitoring

The egress layer manages outbound connections, retry logic, and provider health tracking.

3.5.1 Connection Management

Each provider endpoint maintains a connection pool with configurable maximum connections, keep-alive duration, and TLS session caching. The pool implements HTTP/2 multiplexing where supported, reducing connection overhead for high-throughput providers.

3.5.2 Health Tracking

Provider health is tracked using an exponentially weighted health score $H(p, t)$:

$$H(p,t) = \alpha \cdot h(p,t) + (1-\alpha) \cdot H(p,t-1) \quad (7)$$

where $h(p,t) \in [0,1]$ is the outcome of the most recent request (1 for success, 0 for failure, fractional for degraded performance such as elevated latency), and $\alpha = 0.1$ provides smoothing.

When $H(p,t) < \theta_{\text{open}} = 0.3$, the circuit breaker opens and the provider is temporarily removed from the routing pool. After a configurable cooldown period (default 30 seconds), the circuit enters half-open state, allowing a single probe request. If the probe succeeds, the circuit closes and the provider is restored; otherwise, the cooldown period doubles (exponential backoff up to 5 minutes).

4 Load Balancing

4.1 Adaptive Weight Computation

The load balancer assigns weights to provider endpoints based on observed performance. Let $w_i(t)$ be the weight of provider i at time t:

$$w_i(t) = \frac{H_i(t) \cdot C_i(t)}{\sum_j H_j(t) \cdot C_j(t)} \tag{8}$$

where $H_i(t)$ is the health score and $C_i(t)$ is the remaining rate limit capacity estimated from provider-returned headers:

$$C_i(t) = \min\left(\frac{r_{\text{remaining},i}}{r_{\text{limit},i}}, \frac{t_{\text{remaining},i}}{t_{\text{limit},i}}\right) \tag{9}$$

This formulation naturally routes traffic away from degraded or rate-limited providers without requiring explicit configuration.

4.2 Fallback Chains

Fallback chains define ordered sequences of provider endpoints to attempt for a given request. When the primary provider fails or is unavailable, the gateway transparently retries with the next provider in the chain:

4.2.1 Streaming Failover

Streaming failover presents unique challenges. When a provider fails mid-stream, the gateway must decide whether to: (a) return the partial response, (b) retry from scratch with a different provider, or (c) attempt to continue from the partial output.

We implement strategy (b) with optimization: the gateway buffers streamed tokens and, upon failure, prepends them as an assistant prefix in the retry request. This approach works for providers supporting assistant prefilling (Anthropic, most open-weight model servers) and falls back to full retry otherwise.

Algorithm 1 Fallback Chain Execution

Require: Request q, chain $[p_1, \ldots, p_n]$, retries R

```
errors ← []
for i = 1 to n do
  for r = 1 to R do
    resp ← dispatch(q, p_i)
    if resp.success then
      return resp
    else if resp.retryable then
      sleep(backoff(r))
      errors.append(resp.error)
    else
      errors.append(resp.error)
      break {Non-retryable, try next provider}
    end if
  end for
end for
return AggregateError(errors)
```

4.3 Priority Queues

The gateway implements priority-based request scheduling with four levels:

1. **Critical**: Health checks, auth validation—bypass rate limits.
2. **High**: Interactive user requests—minimize latency.
3. **Normal**: API calls from applications—balance cost and latency.
4. **Low**: Batch processing, embeddings—optimize for cost.

Table 1: Provider pricing model categories

Model	Providers	Unit
Per-token	OpenAI, Anthropic	$/1M tokens
Per-character	Google	$/1M chars
Per-second	Replicate	$/GPU-sec
Reserved	Azure, Bedrock	$/hour
Per-request	Some embeddings	$/request
Tiered	Together AI	Rate-dependent
Free tier	Groq, Cerebras	Quota-limited

Low-priority requests are eligible for delayed dispatch to exploit off-peak pricing windows where available.

5 Cost Tracking and Attribution

5.1 Pricing Model Normalization

Provider pricing models exhibit significant heterogeneity:

The cost engine normalizes all pricing to a per-token basis using provider-specific conversion factors. For character-based pricing, we use the empirical ratio of 1 token $\approx$ 4 characters for English text, calibrated per-model using tokenizer analysis. For time-based pricing, we estimate cost as $c_{\text{time}} = \text{price\_per\_second} \times \text{generation\_time}$.

5.2 Real-Time Cost Computation

For each completed request, the cost engine computes:

$$\text{cost}(q) = c_{\text{in}}(p) \cdot n_{\text{in}} + c_{\text{out}}(p) \cdot n_{\text{out}} + c_{\text{fixed}}(p) \quad (10)$$

where n_{in} and n_{out} are input and output token counts from the provider response's `usage` field, and c_{fixed} captures any per-request fixed costs.

For streaming requests where usage is reported only at stream completion, the gateway estimates running cost from token-counting the streamed content using a provider-matched tokenizer.

5.3 Budget Enforcement

Organizations can set budget limits at multiple granularities:

- **Per-key daily/monthly limits**: Hard caps on API key spend.
- **Per-model limits**: Restrict spend on expensive models.
- **Alerting thresholds**: Notifications at 50%, 80%, 90%, 100% of budget.
- **Automatic downgrade**: When budget thresholds are crossed, automatically route to cheaper model alternatives.

5.4 Cost Attribution

The gateway attributes costs to organizational units through a hierarchical tagging system. Each request can carry tags for organization, project, team, user, and feature. Costs are aggregated along each dimension and exposed through a reporting API:

```
GET /v1/costs?start=2026-02-01
    &end=2026-02-28
    &group_by=project,model
    &filter=team:ml-platform

{
  "total_cost": 14823.47,
  "by_project": {
    "chat-prod": {
      "gpt-4o": 8234.12,
      "claude-sonnet": 3421.89
    },
    "embeddings": {
      "text-embedding-3-large": 1167.46
    }
  }
}
```

Listing 3: Cost attribution query

6 Streaming Architecture

6.1 Server-Sent Events Normalization

The gateway normalizes all streaming protocols to OpenAI-compatible SSE format. This requires handling three distinct upstream protocols:

1. **OpenAI SSE**: `data: {json}` lines with `data: [DONE]` termination.

2. **Anthropic SSE**: Typed events (`event: content_block_delta`) with structured delta payloads.

3. **Google/Vertex**: JSON array streaming or line-delimited JSON.

Each provider adapter implements a streaming transformer that converts provider-specific chunks to normalized `ChatCompletionChunk` objects. The transformer maintains per-stream state to track accumulated content, tool call assembly, and usage statistics.

6.2 Backpressure Management

When downstream clients consume slowly, the gateway must manage backpressure to avoid unbounded memory growth. We implement a bounded buffer with configurable capacity (default 256 chunks). When the buffer fills, the gateway applies backpressure to the upstream provider connection through TCP flow control, pausing reads on the upstream socket.

For providers that do not respect TCP backpressure (those that aggressively push SSE events), the gateway implements an overflow spill-to-disk mechanism with asynchronous replay.

6.3 Multi-Stream Aggregation

For advanced use cases such as parallel model evaluation or ensemble generation, the gateway supports multi-stream aggregation. Multiple provider streams are consumed concurrently, and chunks are interleaved or merged according to a configurable aggregation strategy:

- **First-complete**: Stream the first response to complete, cancel others.
- **Fastest-token**: Forward tokens from whichever stream produces them first.
- **Consensus**: Buffer both streams and return the response with higher self-consistency.

Table 2: Key gateway metrics

Metric	Labels
`llm_request_duration`	provider, model, status
`llm_tokens_total`	provider, model, direction
`llm_cost_usd`	provider, model, org
`llm_cache_hits`	model, cache_tier
`llm_errors_total`	provider, error_type
`llm_ttft_seconds`	provider, model
`llm_provider_health`	provider, endpoint
`llm_queue_depth`	priority_level

7 Observability

7.1 Metrics

The gateway exports Prometheus-compatible metrics at four levels:

Time-to-first-token (TTFT) is tracked separately from total latency, as it directly impacts perceived responsiveness for streaming applications.

7.2 Distributed Tracing

Each request generates an OpenTelemetry trace span with the following structure:

1. `gateway.request`: Root span covering full request lifecycle.

2. `gateway.route`: Routing decision with selected provider and reason.

3. `gateway.cache`: Cache lookup with hit/miss result.

4. `gateway.transform`: Request/response transformation.

5. `gateway.provider`: Upstream provider call with full timing.

6. `gateway.stream`: Streaming duration and chunk count.

Traces are propagated using W3C Trace Context headers, enabling end-to-end tracing from client applications through the gateway to provider endpoints.

7.3 Audit Logging

All requests are logged with configurable verbosity. At the default level, logs include request metadata (model, provider, tokens, cost, latency) without prompt content. At elevated verbosity levels, full prompts and responses are logged with PII redaction applied via configurable regex patterns.

8 Security

8.1 Credential Management

Provider API keys are stored encrypted at rest using AES-256-GCM with keys derived from a master secret via HKDF. The gateway supports integration with external secret managers (HashiCorp Vault, AWS Secrets Manager, Infisical) for key rotation without service restart.

8.2 Request Isolation

Multi-tenant deployments require strict request isolation. The gateway enforces:

- **Key scoping**: API keys are bound to specific organizations with provider access lists.
- **Cache isolation**: Cache entries are scoped by organization to prevent cross-tenant information leakage.
- **Rate limit isolation**: Per-organization rate limits prevent noisy-neighbor effects.
- **Credential isolation**: Each organization can configure its own provider credentials, with the gateway's credentials as fallback.

8.3 Content Filtering

The gateway supports pluggable content filters executed as middleware:

- **PII detection**: Regex and NER-based detection of personally identifiable information with configurable actions (block, redact, log).
- **Prompt injection detection**: Heuristic and classifier-based detection of prompt injection attempts [11].
- **Output filtering**: Post-processing filters for compliance-sensitive content.

9 Implementation

The gateway is implemented in Python using an asynchronous architecture built on `asyncio` and `httpx` for HTTP client operations. The server framework is FastAPI with Uvicorn workers. Key implementation details include:

- **Concurrency model**: Fully asynchronous I/O with no blocking calls. Provider adapters use `httpx.AsyncClient` with connection pooling.
- **Configuration**: YAML-based configuration with environment variable overrides and hot-reload via file watching.
- **Database**: PostgreSQL for persistent state (audit logs, cost records, configuration). Redis for caching and real-time counters.
- **Deployment**: Single Docker image with configurable worker count. Supports horizontal scaling behind any L4/L7 load balancer.

The gateway comprises approximately 28,000 lines of Python code, with provider adapters accounting for 40% of the codebase. Test coverage is 87% with 1,200 unit tests and 340 integration tests.

10 Evaluation

10.1 Experimental Setup

We evaluate the gateway using production traffic from the Hanzo AI platform over a 30-day period (January 2026). The workload comprises:

- 2.8 billion tokens processed (1.1B input, 1.7B output)
- 4.2 million requests across 23 distinct models
- 127 provider endpoints across 14 providers
- Peak throughput: 847 requests/second

Table 3: End-to-end latency overhead (milliseconds)

Component	p50	p95	p99	Max
Auth + Rate Limit	0.3	0.8	1.2	4.1
Routing Decision	0.1	0.4	0.7	2.3
Cache Lookup	0.4	1.1	2.8	8.7
Transform (req)	0.2	0.6	1.1	3.4
Transform (resp)	0.3	0.7	1.3	4.2
Total Overhead	**1.3**	**3.6**	**7.1**	**22.7**

10.2 Latency

The gateway adds a median of 1.3ms of overhead per request, which is negligible compared to typical LLM inference latencies of 500ms–30s. The p99 overhead of 7.1ms is dominated by cache lookup in the Redis tier.

Intelligent routing reduces median end-to-end latency by 23% compared to fixed-provider routing by directing requests to the fastest available provider. The improvement is most pronounced during provider degradation events, where failover avoids multi-second timeouts.

10.3 Cache Performance

Over the evaluation period, the semantic cache achieved:

- **Hit rate**: 34.2% of all requests served from cache.
- **Precision**: 97.2% of cache hits returned responses equivalent to fresh provider responses (validated on a 10,000-request sample).
- **Latency reduction**: Cache hits served in median 1.8ms vs. 1,240ms for cache misses (690× speedup).
- **Cost savings**: Cache hits eliminated $47,200 in provider costs over the evaluation period.

Cache hit rates vary significantly by use case: embedding requests achieve 71% hit rate (high repetition), while creative generation requests achieve only 8% (high temperature, unique prompts).

Table 4: Monthly cost comparison (USD)

Strategy	Cost	Savings	Qual.
Single provider (OpenAI)	183,400	—	1.00
Manual multi-provider	152,100	17%	0.98
Gateway: cost-optimized	126,500	31%	0.96
Gateway: quality-first	171,200	7%	1.02
Gateway: balanced	141,300	23%	0.99

Table 5: Provider failover events during evaluation

Event Type	Count	Avg Duration	Requests Saved
Rate limit	1,247	12s	34,200
5xx error	89	4.2min	12,100
Timeout	312	8s	8,900
Full outage	3	47min	28,400

10.4 Cost Optimization

The cost-optimized strategy reduces costs by 31% with a 4% quality degradation as measured by a composite benchmark of MMLU, HumanEval, and MT-Bench scores. The balanced strategy achieves 23% cost reduction with negligible quality impact.

10.5 Availability

During the 30-day evaluation, the gateway achieved 99.97% availability (8 minutes of cumulative downtime across two rolling restart events). Provider-level availability varied from 99.2% (worst single provider) to 99.99% (best). The gateway's multi-provider failover elevated effective availability from the worst single-provider level to the aggregate level.

10.6 Throughput

Under load testing with synthetic traffic, the gateway sustains:

- 2,400 requests/second with non-streaming responses (single node, 8 Uvicorn workers).
- 1,800 concurrent streaming connections with sub-5ms chunk relay latency.

- Linear horizontal scaling up to 12 nodes (tested), with shared Redis state for cache and rate limits.

11 Discussion

11.1 Limitations

Semantic cache precision. While 97.2% precision is high, the 2.8% false positive rate means approximately 1 in 36 cache hits returns a subtly incorrect response. For safety-critical applications, we recommend disabling semantic caching or reducing the similarity threshold.

Provider-specific features. Some provider-specific features (e.g., Anthropic's extended thinking, OpenAI's structured outputs with `strict: true`) require pass-through handling that bypasses normalization. The gateway supports a `provider_params` escape hatch for such cases, but this breaks provider-agnostic routing.

Real-time pricing. Cost tracking relies on a pricing database that must be manually updated when providers change prices. We are developing automated price scraping and notification systems to reduce this operational burden.

11.2 Future Work

- **Learned routing**: Training a lightweight model to predict optimal provider selection based on prompt features, replacing the heuristic scoring functions.
- **Speculative execution**: Dispatching requests to multiple providers simultaneously and returning the first successful response, trading cost for latency.
- **Prompt optimization**: Automatically reformatting prompts for provider-specific strengths (e.g., adding chain-of-thought markers for models that benefit from them).
- **Federated deployment**: Multi-region gateway instances with global routing for compliance and latency optimization.

12 Conclusion

The Hanzo LLM Gateway demonstrates that a well-designed proxy layer can substantially reduce the complexity, cost, and risk of multi-provider LLM deployments. By normalizing 100+ provider APIs behind a single interface, implementing intelligent routing with formal optimization objectives, and providing semantic caching with high precision, the gateway achieves 23% latency reduction, 31% cost savings, and 99.97% availability in production. The open-source release enables organizations to deploy multi-provider LLM infrastructure without vendor lock-in.

References

[1] T. Brown, B. Mann, N. Ryder, et al. Language models are few-shot learners. In *NeurIPS*, 2020.

[2] R. Bommasani, D. Hudson, E. Adeli, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2022.

[3] AI Infrastructure Alliance. State of AI infrastructure 2025: Multi-provider deployment patterns. Technical report, 2025.

[4] W. Kwon, Z. Li, S. Zhuang, et al. Efficient memory management for large language model serving with PagedAttention. In *SOSP*, 2023.

[5] HuggingFace. Text generation inference: A Rust, Python and gRPC server for text generation. `https://github.com/huggingface/text-generation-inference`, 2023.

[6] BerriAI. LiteLLM: Call all LLM APIs using the OpenAI format. `https://github.com/BerriAI/litellm`, 2023.

[7] Portkey. Portkey: Control panel for AI apps. `https://portkey.ai`, 2024.

[8] Martian. Model router: Automatic model selection for optimal cost and quality. Technical report, Martian AI, 2024.

[9] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[10] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[11] F. Perez and I. Ribas. Ignore this title and HackAPrompt: Exposing systemic weaknesses of LLMs through a global-scale prompt hacking competition. *arXiv preprint arXiv:2311.16119*, 2022.

[12] OpenAI. OpenAI API reference. `https://platform.openai.com/docs/api-reference`, 2024.

[13] Anthropic. Anthropic API reference. `https://docs.anthropic.com/en/api`, 2024.

[14] Google. Gemini API documentation. `https://ai.google.dev/docs`, 2024.

[15] M. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2nd edition, 2018.

[16] B. Burns. *Designing Distributed Systems*. O'Reilly Media, 2018.

[17] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[18] S. Vargaftik, R. Ben-Basat, A. Szpigel, and G. Mendelson. RADE: Resource-efficient supervised anomaly detection using decision tree encoding. In *MLSys*, 2022.

[19] L. Chen, M. Zaharia, and J. Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[20] A. Jiang, A. Sablayrolles, A. Roux, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[21] H. Touvron, L. Martin, K. Stone, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[22] Gemma Team. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[23] L. Zheng, W.-L. Chiang, Y. Sheng, et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *NeurIPS*, 2023.

[24] D. Hendrycks, C. Burns, S. Basart, et al. Measuring massive multitask language understanding. In *ICLR*, 2021.

[25] M. Chen, J. Tworek, H. Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.