Intelligent LLM Routing: Multi-Provider Abstraction with Adaptive Complexity Classification

Kaman AI Research

March 2026

Abstract

Enterprise AI platforms must integrate multiple large language model (LLM) providers—OpenAI, Anthropic, Google, Groq, Azure, AWS Bedrock, and others—each offering distinct models with varying cost, latency, capability, and availability characteristics. The prevailing approach of hardcoding a single provider creates vendor lock-in, exposes organizations to single points of failure, and forecloses the cost optimization opportunities inherent in a heterogeneous model landscape. We present a *multi-provider routing architecture* that abstracts the provider layer behind an OpenAI-compatible API surface, enabling transparent model substitution, automatic failover, and—critically—*adaptive complexity classification* that routes prompts to the most cost-effective model capable of handling them. The system introduces a multi-scope model management framework (Global, Client, Role, Private) with priority-based resolution, provider-agnostic health monitoring with circuit breaker patterns, and a lightweight neural classifier that estimates prompt complexity from lexical, syntactic, and domain features. In production deployment across enterprise workloads, the architecture reduces LLM inference costs by 35–55% while improving effective availability from 99.5% to 99.95% through automatic failover chains. We describe the system design, the complexity classification approach, and present empirical evaluation across latency, cost, and reliability dimensions.

1 Introduction

The rapid proliferation of large language model providers has created an unprecedented diversity of inference options for enterprise AI platforms. As of early 2026, organizations can choose from over forty foundation models across a dozen providers, each presenting distinct tradeoffs along the dimensions of capability, latency, cost, context window size, and specialized competencies such as code generation, multilingual understanding, or structured output compliance.

This diversity is simultaneously an opportunity and an operational burden. The opportunity lies in *model-task matching*: not every prompt requires the most capable (and expensive) model. A simple formatting request, a routine data lookup, or a straightforward classification task can be handled by a smaller, faster, cheaper model with no degradation in output quality. Conversely, complex multi-step reasoning, nuanced creative writing, or tasks requiring large context windows demand frontier-class models.

The burden manifests in three interconnected challenges:

1. **Provider fragmentation.** Each provider exposes a different API format, authentication mechanism, error taxonomy, and streaming protocol. Supporting P providers requires P distinct integration paths, each with its own retry logic, rate limiting, and credential management.
2. **Availability risk.** Provider outages, rate limit exhaustion, and regional failures are not exceptional events—they are routine operational realities. Organizations that depend on a single provider experience cascading failures when that provider becomes unavailable, even if equivalent capacity exists elsewhere.
3. **Cost opacity.** Without centralized routing and metering, organizations lack visibility into which models are being used for which tasks, making it impossible to optimize spending. A common anti-pattern is routing all requests to the most capable model "just to be safe," resulting in 3–10x cost overruns relative to an optimized routing strategy.

We present the *Model Proxy*, a stateless microservice that addresses these challenges through three mechanisms: (1) a provider abstraction layer that normalizes heterogeneous APIs behind a single OpenAI-compatible interface; (2) an adaptive complexity classifier that routes prompts to cost-appropriate models; and (3) a multi-scope model management framework that enables fine-grained control over model availability across organizational hierarchies.

The contributions of this paper are:

- A provider abstraction architecture supporting nine providers with automatic capability detection and streaming normalization (Section 3).
- An adaptive complexity classification system that routes prompts based on learned features, reducing costs without degrading output quality (Section 4).
- A multi-scope model management framework with priority-based resolution across organizational hierarchies (Section 5).
- Automatic fallback chains with circuit breaker patterns that improve effective availability to 99.95% (Section 6).
- Empirical evaluation demonstrating 35–55% cost reduction and sub-100ms routing overhead across production workloads (Section 8).

2 System Architecture

The Model Proxy is implemented as a stateless FastAPI microservice that sits between the application layer (agent framework, frontend clients, workflow engine) and the LLM providers. Its position in the stack is analogous to a reverse proxy or API gateway, but specialized for LLM inference traffic.

2.1 Design Principles

The architecture is governed by four principles:

OpenAI compatibility. The proxy exposes `/v1/chat/completions` and `/v1/embeddings` endpoints that conform to the OpenAI API specification. Any client that can call OpenAI can call the proxy with zero code changes—only the base URL and authentication header differ. This design decision eliminates migration friction and enables gradual adoption.

Statelessness. The proxy maintains no per-request state between invocations. Model configurations, scope resolution rules, and provider credentials are loaded from a shared PostgreSQL database and cached with configurable TTLs. This enables horizontal scaling behind a load balancer without session affinity requirements.

Provider opacity. From the client's perspective, the provider that ultimately serves a request is an implementation detail. The client requests a *logical model name* (e.g., `gpt-4`, `claude-3`), and the proxy resolves this to a concrete provider endpoint based on availability, scope rules, and routing policy. The response is normalized to OpenAI format regardless of the underlying provider.

Zero-downtime extensibility. Adding a new provider requires implementing a single interface (Section 3) and registering the provider in the model database. No proxy restart or client changes are needed.

2.2 Request Flow

A typical request traverses the following pipeline:

1. **Authentication.** The proxy validates the incoming token against the Kaman Auth Service, extracting user identity, client (organization) membership, and role assignments (Section 7).
2. **Scope resolution.** The requested model name is resolved against the multi-scope model registry, yielding a concrete model configuration with provider, API key, and parameters (Section 5).
3. **Complexity classification.** The prompt is analyzed by the complexity classifier, which may override the model selection if routing policies are active (Section 4).
4. **Provider dispatch.** The request is translated to the target provider's native format and dispatched via the appropriate provider adapter.
5. **Response normalization.** The provider's response (streaming or batch) is translated back to OpenAI format and returned to the client.
6. **Fallback handling.** If the primary provider fails, the fallback chain is activated, repeating steps 4–5 with alternative providers (Section 6).

2.3 API Surface

The proxy exposes two primary endpoints:

```
POST /v1/chat/completions
  - model: str (logical model name)
  - messages: list[Message]
  - temperature, max_tokens, ...
  - stream: bool
  - tools: list[Tool] (optional)
  - tool_choice: str (optional)

POST /v1/embeddings
  - model: str (embedding model name)
  - input: str | list[str]
```

Both endpoints accept standard OpenAI request bodies and return OpenAI-compatible responses, including streaming via Server-Sent Events (SSE) with `data: [DONE]` termination.

Administrative endpoints for model management, health monitoring, and usage analytics are exposed on a separate internal port with role-based access controls.

3 Provider Abstraction Layer

The provider abstraction layer normalizes the heterogeneous API surfaces of nine LLM providers behind a uniform interface. Each provider is implemented as a concrete class that extends a base provider interface, handling the translation between the proxy's internal request format and the provider's native API.

3.1 Base Provider Interface

The interface defines three core operations:

- `chat_completion(request)` — Synchronous (batch) chat completion. Accepts a normalized request object and returns a normalized response.
- `chat_completion_stream(request)` — Streaming chat completion. Returns an async generator yielding normalized SSE chunks.
- `embeddings(request)` — Text embedding. Returns a list of dense vectors.

Each method accepts a provider-agnostic request object and is responsible for: (a) translating the request to the provider's native format, (b) dispatching the HTTP call with appropriate authentication, (c) translating the response back to the normalized format, and (d) raising standardized exceptions for error conditions.

3.2 Supported Providers

Table 1 summarizes the nine supported providers and their distinguishing characteristics.

Table 1: Supported LLM providers and key capabilities.

Provider	Notable Models	Stream	Tools
OpenAI	GPT-4o, o3, o4-mini	Yes	Yes
Anthropic	Claude 4, Opus, Sonnet	Yes	Yes
Groq	Llama 3, Mixtral	Yes	Yes
Azure	GPT-4o (hosted)	Yes	Yes
Bedrock	Claude, Titan, Llama	Yes	Yes
Ollama	Local open models	Yes	Partial
vLLM	Self-hosted endpoints	Yes	Partial
xAI	Grok-2, Grok-3	Yes	Yes
Cerebras	Inference-optimized	Yes	Yes

3.3 Capability Detection

Not all providers support all features. Function calling (tool use), vision inputs, JSON mode, and long context windows are capabilities that vary across providers and even across models within the same provider.

The proxy maintains a capability registry that maps each model to its supported feature set. When a request includes features not supported by the target model (e.g., tool calls to a model that lacks function calling), the proxy can either reject the request with a descriptive error, attempt a graceful degradation (e.g., serializing tool definitions into the system prompt), or route to an alternative model that supports the required capability.

Capability information is stored in the model database alongside provider configuration, enabling administrators to update capability mappings without code changes as providers evolve their offerings.

3.4 Streaming Normalization

Streaming protocols differ significantly across providers. OpenAI uses SSE with `data:` prefixed JSON chunks and a `[DONE]` sentinel. Anthropic uses a distinct event-based SSE protocol with typed events (`content_block_delta`, `message_stop`). Some providers return NDJSON or chunked transfer encoding without SSE framing.

The provider abstraction layer normalizes all streaming protocols to the OpenAI SSE format. Each provider's stream adapter yields normalized chunk objects that the proxy serializes into the standard `data: {...}` format. This normalization is transparent to clients, which receive identically formatted streams regardless of the underlying provider.

3.5 Error Taxonomy

Provider errors are mapped to a standardized error taxonomy:

- **Rate limit** (HTTP 429) — triggers backoff and potential fallback.
- **Authentication** (HTTP 401/403) — logged as configuration error, no retry.
- **Model unavailable** (HTTP 503) — triggers immediate fallback.
- **Context overflow** (HTTP 400 with specific codes) — logged with context size metrics.
- **Timeout** — triggers fallback after configurable deadline.

This taxonomy enables the fallback engine (Section 6) to make informed decisions about whether an error is transient (retry the same provider), structural (switch providers), or permanent (return error to client).

4 Adaptive Complexity Classification

The central insight motivating intelligent routing is that prompt complexity is highly variable across enterprise workloads. Analysis of production traffic reveals that a significant fraction of requests—simple lookups, formatting tasks, straightforward classifications—can be served by smaller, faster, cheaper models with no measurable quality degradation. Routing these requests to frontier models wastes both cost and latency.

4.1 Problem Formulation

Given a prompt x (comprising system message, conversation history, and user message), a set of available models $\mathcal{M} = \{m_1, m_2, \ldots, m_K\}$, and a quality threshold τ, the routing objective is:

$$m^* = \arg\min_{m \in \mathcal{M}} \text{cost}(m) \quad \text{s.t.} \quad Q(m, x) \geq \tau \tag{1}$$

where $Q(m, x)$ denotes the expected output quality of model m on prompt x, and $\text{cost}(m)$ represents the per-token inference cost. The challenge is estimating $Q(m, x)$ *before* inference, using only the prompt features.

Direct estimation of Q is intractable—it would require running inference on all candidate models. Instead, we decompose the problem by introducing a *complexity function* $\phi(x) \in [0, 1]$ that maps prompts to a scalar complexity score, and a *model capability threshold* θ_m for each model m:

$$Q(m, x) \geq \tau \iff \phi(x) \leq \theta_m \tag{2}$$

Under this formulation, the routing decision reduces to computing $\phi(x)$ and selecting the cheapest model whose capability threshold exceeds the prompt's complexity.

4.2 Feature Extraction

The complexity classifier operates on a feature vector $\mathbf{f}(x) \in \mathbb{R}^d$ extracted from the prompt. We employ a multi-signal feature extraction pipeline that captures lexical, structural, and domain characteristics:

Lexical features. Token count, vocabulary diversity (type-token ratio), average word length, and presence of specialized terminology. These features capture surface-level complexity: longer prompts with diverse vocabulary and domain jargon tend to require more capable models.

Structural features. Sentence count, question density, presence of nested conditionals ("if…then…otherwise"), enumeration markers, and multi-part request indicators. Multi-step instructions and conditional logic are strong signals of reasoning complexity.

Domain signals. Presence of code blocks, mathematical notation, tabular data, JSON/XML structures, and domain-specific entity patterns. Code generation and mathematical reasoning are consistently more demanding than natural language tasks.

Conversational features. History length, turn count, presence of tool call results in context, and accumulated context size. Long multi-turn conversations with tool interactions typically require stronger models to maintain coherence.

Task type indicators. Imperative verbs and their associated complexity classes: "format," "list," "translate" signal simpler tasks; "analyze," "compare," "design," "evaluate" signal complex reasoning.

The feature vector is computed in constant time with respect to the vocabulary size, ensuring that the classification overhead remains negligible relative to inference latency.

4.3 Classification Architecture

The complexity score $\phi(x)$ is computed by a lightweight neural classifier trained on production data. The architecture is deliberately compact—the classifier must add negligible latency to the routing path.

The model maps the feature vector $\mathbf{f}(x)$ through a small number of layers with nonlinear activations, producing a scalar output in $[0, 1]$ via a sigmoid function:

$$\phi(x) = \sigma\big(g(\mathbf{f}(x); \Theta)\big) \tag{3}$$

where $g(\cdot;\Theta)$ denotes the parameterized network and σ is the sigmoid function. The output is interpreted as a complexity score: values near 0 indicate simple prompts suitable for the cheapest models, while values near 1 indicate complex prompts requiring frontier-class capabilities.

Training data is derived from production logs using a retrospective labeling strategy. For each historical request, we compute a quality signal based on downstream outcomes: user satisfaction indicators, retry rates, task completion signals, and explicit feedback where available. Prompts where smaller models produced satisfactory results are labeled as low-complexity; prompts where only frontier models succeeded are labeled as high-complexity.

The classifier is retrained periodically as new production data accumulates and as the model landscape evolves. We do not disclose the exact architecture depth, width, regularization strategy, or training hyperparameters, as these represent proprietary optimizations tuned to enterprise workload distributions.

4.4 Routing Policy

The complexity score $\phi(x)$ is mapped to a model selection through a *routing policy* that defines complexity tiers:

$$m^*(x) = \begin{cases} m_{\text{economy}} & \text{if } \phi(x) < \alpha \\ m_{\text{standard}} & \text{if } \alpha \leq \phi(x) < \beta \\ m_{\text{frontier}} & \text{if } \phi(x) \geq \beta \end{cases} \quad (4)$$

The thresholds α and β are calibrated per deployment based on the organization's cost-quality tradeoff preferences. A conservative deployment (prioritizing quality) uses lower thresholds, routing more traffic to capable models. An aggressive deployment (prioritizing cost) uses higher thresholds, routing more traffic to economy models.

The routing policy is applied *after* scope resolution (Section 5), ensuring that the classifier only routes to models the user is authorized to access. Administrators can disable complexity-based routing entirely, in which case the proxy acts as a pure passthrough to the scope-resolved model.

4.5 Override Mechanisms

Two mechanisms allow bypassing the complexity classifier:

Explicit model pinning. When a client specifies a model with an explicit provider prefix (e.g., `anthropic/claude-4-opus`), the classifier is bypassed and the request is routed directly to the specified model. This enables power users and automated pipelines to control model selection when task requirements are known a priori.

Minimum model floor. Administrators can configure a minimum model tier per application or per agent. This ensures that certain high-stakes applications (e.g., customer-facing agents, financial analysis) always use at least a standard-tier model, regardless of the classifier's assessment.

5 Multi-Scope Model Management

Enterprise deployments require fine-grained control over which models are available to which users, teams, and organizations. A single global model catalog is insufficient: different organizations have different provider contracts, different teams have different cost budgets, and individual users may have personal model preferences or experimental access to preview models.

5.1 Scope Hierarchy

The Model Proxy implements a four-level scope hierarchy for model management:

1. **Global scope.** Models available to all users across all organizations. These represent the platform's baseline offering—generally available models from major providers. Global models are managed by platform administrators.
2. **Client scope.** Models available to all users within a specific organization ("client" in Kaman terminology). These represent organization-specific provider contracts, custom fine-tuned models, or self-hosted inference endpoints. Client models are managed by organization administrators.
3. **Role scope.** Models available to users with a specific role within an organization. These enable capability-based access: an "engineering" role might have access to code-specialized models, while a "marketing" role has access to content-generation models. Role models are managed by organization administrators.
4. **Private scope.** Models available only to a specific user. These represent personal API keys (e.g., a developer's own OpenAI key), experimental model access, or user-specific fine-tuned

models. Private models are self-managed by individual users.

5.2 Priority Resolution

When a client requests a model by logical name (e.g., `gpt-4`), the proxy resolves the name against all applicable scopes and selects the highest-priority match:

$$\text{resolve}(\text{name}, u) = \text{first}\big(\mathcal{M}^{u}_{\text{priv}},\ \mathcal{M}^{r(u)}_{\text{role}},\ \mathcal{M}^{c(u)}_{\text{client}},\ \mathcal{M}_{\text{global}}\big) \tag{5}$$

where u is the user, $r(u)$ is the user's role set, $c(u)$ is the user's organization, and $\text{first}(\cdot)$ returns the first scope containing a model with the requested name.

This resolution order ensures that more specific configurations override less specific ones:

- A user's private `gpt-4` configuration (using their personal API key) overrides the organization's configuration.
- An organization's `gpt-4` configuration (routed through their Azure deployment) overrides the global default (routed through OpenAI directly).
- A role-specific model (e.g., `claude-code` available only to engineering roles) is invisible to users without that role.

5.3 Model Database Schema

Model configurations are stored in a dedicated `llm_models` table with the following key attributes:

- **Identity:** Logical name, display name, provider, provider-specific model ID.
- **Scope:** Scope level (global/client/role/private), associated user/client/role IDs.
- **Capabilities:** Supported features (streaming, function calling, vision, JSON mode), context window size, maximum output tokens.
- **Economics:** Cost per million input tokens, cost per million output tokens.
- **Defaults:** Default temperature, top-p, and other inference parameters.
- **Status:** Active/inactive flag, priority within scope.

The schema supports multiple models with the same logical name across different scopes, enabling the priority resolution described above. Capability and cost metadata is used by the complexity classifier (Section 4) to inform routing decisions.

5.4 Capability-Based Filtering

When a request includes features that require specific model capabilities (e.g., tool calls, vision inputs, structured output), the scope resolution process filters candidate models by capability before applying priority resolution. This ensures that the resolved model can actually handle the request, even if a higher-priority model in a more specific scope lacks the required capability.

For example, if a user's private model does not support function calling but the request includes tool definitions, the resolver falls through to the next scope that offers a function-calling-capable model. This capability-aware fallthrough is transparent to the client and ensures correct behavior without requiring callers to understand provider-level feature availability.

6 Automatic Fallback Chains

Provider failures—rate limits, outages, timeouts, regional disruptions—are routine operational events in multi-provider environments. The Model Proxy implements automatic fallback chains that transparently reroute failed requests to alternative providers, maximizing effective availability.

6.1 Fallback Chain Definition

Each logical model name is associated with an ordered list of provider endpoints that can serve equivalent functionality. Fallback chains are defined based on model family equivalence:

- **Claude family:** Anthropic API → AWS Bedrock → Google Vertex AI.
- **GPT family:** OpenAI API → Azure OpenAI → Groq (for compatible models).
- **Open models:** Primary self-hosted (vLLM) → Secondary self-hosted (Ollama) → Groq → Cerebras.

The fallback order is configurable per model and per organization, allowing administrators to reflect their provider preferences, contract terms, and data residency requirements.

6.2 Health Monitoring

The proxy continuously monitors provider health through two mechanisms:

Passive monitoring. Every inference request produces a health signal: success, latency, error type, and error code. These signals are aggregated into per-provider, per-model health metrics with configurable rolling windows. Metrics include success rate, median latency, p95 latency, and error rate by category.

Active probing. A background process periodically sends lightweight probe requests to each provider endpoint. Probes use minimal-cost prompts and are designed to verify connectivity, authentication, and basic inference capability without incurring significant cost.

6.3 Circuit Breaker Pattern

When a provider's error rate exceeds a configurable threshold within a monitoring window, the circuit breaker *opens*, temporarily removing the provider from the active routing pool. The circuit breaker progresses through three states:

1. **Closed** (normal operation): Requests are routed to the provider. Errors are counted against the threshold.
2. **Open** (provider failing): No requests are routed to the provider. A recovery timer begins.
3. **Half-open** (recovery testing): After the timer expires, a single probe request is sent. If it succeeds, the circuit closes; if it fails, the circuit reopens with a longer timer following an exponential backoff schedule.

The circuit breaker prevents cascading failures: when a provider is experiencing elevated errors, continued request routing increases latency (due to timeouts) and cost (due to wasted tokens on partial responses). By removing the failing provider from the pool, the proxy immediately shifts traffic to healthy alternatives.

6.4 Fallback Latency

Table 2 presents observed fallback recovery times across common failure scenarios.

Table 2: Fallback recovery times by failure type.

Failure Type	Detect (ms)	Switch (ms)	Total (ms)
Rate limit (429)	5	50	55
Auth error (401)	5	50	55
Timeout (30s)	30,000	50	30,050
Connection refused	200	50	250
Circuit open	0	50	50

For non-timeout failures, fallback adds less than 100ms to request latency. Timeout-triggered fallbacks are inherently expensive, as the system must wait for the timeout deadline before switching. To mitigate this, the proxy supports *speculative execution*: for high-priority requests, the proxy can dispatch to both the primary and first fallback provider simultaneously, returning whichever responds first. This trades cost for latency, and is configurable per model tier.

7 Authentication Integration

The Model Proxy integrates with the Kaman Auth Service for request authentication and authorization. This integration serves two purposes: validating that the requesting entity is authorized to use the proxy, and extracting identity information needed for scope resolution.

7.1 Token Validation

Every incoming request must include a Kaman user token in the `Authorization` header. The proxy validates this token by calling the Auth Service's `/auth/userinfo` endpoint, which returns:

- User identity (ID, email, display name).
- Organization membership (client ID, client name).
- Role assignments (list of role IDs and names).
- Token validity and expiration metadata.

Validation results are cached with a short TTL to avoid per-request round trips to the Auth Service. The cache is keyed by token hash and invalidated on expiration or explicit revocation signals.

7.2 Scope Extraction

The user identity, organization, and role information extracted during authentication are passed directly to the scope resolution engine (Section 5). This ensures that model access is always evaluated against the user's current identity state—if a user's role changes or organization membership is updated, the next request will reflect the updated model availability without requiring token reissuance.

7.3 Usage Attribution

Every inference request is logged with the authenticated user's identity, enabling per-user, per-organization, and per-role usage tracking. This

data feeds into billing systems, usage dashboards, capacity planning, and the complexity classifier's training pipeline. Logged metadata includes model name, provider, input/output token counts, latency, and routing decisions (whether fallback was triggered, whether the complexity classifier overrode the requested model).

The proxy does not store or log prompt content—only request metadata is retained for analytics. This design reflects a defense-in-depth approach to data privacy: even if the analytics store is compromised, no sensitive prompt data is exposed.

8 Evaluation

We evaluate the Model Proxy across three dimensions: inference latency overhead, cost reduction from intelligent routing, and availability improvement from automatic fallback. All measurements are drawn from production deployment over a rolling 90-day window.

8.1 Latency Overhead

Table 3 reports the latency overhead introduced by the proxy layer, measured as the difference between end-to-end request latency through the proxy and direct provider latency for the same request.

Table 3: Proxy latency overhead by component (median / p95).

Component	Median (ms)	p95 (ms)
Auth validation (cached)	1	3
Scope resolution	2	5
Complexity classification	3	8
Request translation	1	2
Response normalization	1	3
Total overhead	**8**	**21**

The total proxy overhead of 8ms (median) is negligible relative to typical LLM inference latencies, which range from 200ms to 30 seconds depending on model, prompt length, and output length. Even at the 95th percentile, the 21ms overhead represents less than 1% of a typical inference call.

The complexity classifier contributes 3ms at the median, dominated by feature extraction from the prompt text. The classification inference itself (the forward pass through the neural network) accounts for less than 0.5ms on commodity hardware.

8.2 Cost Reduction

We analyze cost reduction from complexity-based routing across a representative sample of production traffic. Table 4 compares monthly costs under three routing strategies: static (all requests to a frontier model), manual tiering (developer-assigned model per application endpoint), and adaptive (complexity classifier).

Table 4: Monthly LLM cost comparison across routing strategies (normalized to static routing = 100).

Workload Type	Static	Manual	Adaptive
General chat	100	72	48
Data analysis	100	85	62
Code generation	100	90	78
Document processing	100	65	41
Mixed enterprise	100	75	55

Adaptive routing achieves a 45% cost reduction on the mixed enterprise workload compared to static routing, and a 27% reduction compared to manual tiering. The greatest savings are observed on document processing (59% reduction), where many requests involve simple formatting and extraction tasks that the classifier correctly routes to economy-tier models. Code generation shows the smallest savings (22% reduction), as the classifier conservatively routes most coding tasks to frontier models—a calibration choice reflecting the higher cost of errors in generated code.

8.3 Quality Preservation

Cost reduction is meaningful only if output quality is preserved. We measure quality through three proxy metrics derived from production signals:

- **Retry rate:** The fraction of requests where the user or application retried with the same or similar prompt, indicating dissatisfaction with the initial response.
- **Escalation rate:** The fraction of adaptively-routed requests that were subsequently resubmitted with an explicit frontier model pin, indicating that the routing decision was suboptimal.
- **Task completion:** For workflow-embedded requests, the fraction that completed without requiring human intervention or error recovery.

Across production workloads, the retry rate under adaptive routing is within 2 percentage points of the static routing baseline. The escalation rate remains below 3%, and task completion rates are statistically indistinguishable between static and adaptive routing at the 95% confidence level. These metrics confirm that the complexity classifier achieves cost reduction without meaningful quality degradation.

8.4 Availability Improvement

We measure effective availability as the fraction of requests that receive a successful response (HTTP 200) within the timeout deadline.

Under single-provider deployment, availability is bounded by the provider's own SLA and operational stability. Observed single-provider availability across major providers ranges from 99.2% to 99.8% over a rolling 30-day window, with periodic degradation events lasting minutes to hours.

With automatic fallback chains, the proxy's effective availability reaches 99.95% over the same observation period. The improvement stems from the observation that provider failures are largely uncorrelated: when one provider experiences elevated error rates, alternative providers in the fallback chain are typically unaffected. Formally, if each provider has independent availability A_i, the effective availability of a chain of n providers is:

$$A_{\text{eff}} = 1 - \prod_{i=1}^{n}(1 - A_i) \tag{6}$$

For a chain of three providers each with $A_i = 0.995$, $A_{\text{eff}} \approx 0.999999875$—effectively five nines. In practice, failures are not perfectly independent (e.g., upstream infrastructure failures can affect multiple providers), but the observed 99.95% availability substantially exceeds any single provider's track record.

8.5 Scaling Characteristics

The proxy's stateless design enables linear horizontal scaling. In load testing, a single proxy instance sustains approximately 500 concurrent inference requests with sub-20ms routing overhead. Adding instances behind a load balancer scales throughput linearly, as there is no shared mutable state requiring coordination between instances.

The primary scaling consideration is the auth token cache: under high request rates with diverse user populations, cache miss rates increase, causing more round trips to the Auth Service. This is mitigated by tuning the cache TTL and ensuring the Auth Service is itself horizontally scaled.

9 Related Work

LiteLLM [1] provides an open-source Python library that wraps multiple LLM providers behind an OpenAI-compatible interface. Our work extends this concept from a client-side library to a managed service with centralized authentication, multi-scope model management, and adaptive routing—capabilities that a library embedded in each calling application cannot provide in a multi-tenant environment. The Model Proxy initially used LiteLLM as its provider layer before evolving to a custom provider abstraction for finer-grained control over streaming, error handling, and capability detection.

OpenRouter [2] operates a hosted LLM routing service with model selection and fallback. While conceptually similar, OpenRouter is a third-party SaaS product, requiring organizations to route sensitive prompts through external infrastructure. Our proxy is self-hosted within the organization's network boundary, ensuring that prompt data never traverses external systems.

Portkey [3] provides an AI gateway with load balancing, request caching, and observability for LLM APIs. Portkey focuses on operational reliability rather than intelligent routing; it does not implement complexity-based model selection. The operational patterns Portkey describes (circuit breakers, retry logic, request caching) informed our fallback engine design.

FrugalGPT [4] proposes a cascade strategy where prompts are first sent to cheaper models, and only escalated to expensive models if the initial response is judged unsatisfactory. While effective for batch processing, this approach doubles latency for complex prompts (which must fail on the cheap model before being escalated) and requires a quality judge that itself consumes inference budget. Our pre-dispatch classification avoids this latency penalty by routing to the appropriate tier on the first attempt.

RouteLLM [5] presents an open-source framework for LLM routing with learned routers that

dynamically select between a stronger and weaker model pair. Their router architectures (matrix factorization, BERT-based classifiers, causal LLM routers) informed our feature engineering approach, though our production classifier is optimized for lower inference latency and operates over a wider set of routing targets with organizational context that external routers cannot observe.

Martian [6] introduces a learned router that selects among LLM providers based on prompt characteristics. Their approach is closest to our complexity classifier in spirit, but operates as an external routing service rather than an integrated component of an enterprise platform with access to user, role, and organizational context.

Mixture-of-Agents [7] explores using multiple LLMs collaboratively, with outputs from weaker models feeding into stronger models as auxiliary context. This architecture targets quality improvement through model composition rather than cost optimization through intelligent selection, and is orthogonal to our routing approach.

10 Conclusion

We have presented the Model Proxy, a multi-provider LLM routing architecture that addresses the interconnected challenges of provider fragmentation, availability risk, and cost opacity in enterprise AI deployments. The system's three principal contributions—provider abstraction, adaptive complexity classification, and multi-scope model management—work in concert to deliver a unified inference layer that is transparent to clients, resilient to provider failures, and cost-optimized across heterogeneous workloads.

The provider abstraction layer demonstrates that nine distinct LLM providers can be normalized behind a single OpenAI-compatible API surface with less than 10ms of median routing overhead. This normalization eliminates the integration burden of multi-provider support and enables automatic fallback chains that improve effective availability from 99.5% to 99.95%.

The adaptive complexity classifier demonstrates that a lightweight learned model, operating on lexical, structural, and domain features extracted from the prompt, can route 45–60% of enterprise LLM traffic to cheaper models without measurable quality degradation. The classifier adds only 3ms of median latency—negligible relative to inference times—while delivering 35–55% cost reduction across mixed enterprise workloads.

The multi-scope model management framework demonstrates that fine-grained, hierarchical control over model availability is essential for enterprise deployments. The four-scope hierarchy (Global, Client, Role, Private) with priority-based resolution enables organizations to maintain centralized governance while accommodating the diverse needs of individual teams and users.

Several directions for future work are under investigation:

Continuous classifier adaptation. The current classifier is retrained periodically on accumulated production data. An online learning variant that continuously adapts to shifting workload distributions and evolving model capabilities would improve routing accuracy without manual retraining cycles.

Cost-aware context management. Integrating the routing engine with upstream context management (prompt compression, history truncation, retrieval-augmented generation filtering) could compound cost savings by minimizing input token counts before routing.

Multi-objective routing. The current classifier optimizes for cost subject to a quality threshold. A multi-objective formulation that jointly optimizes cost, latency, and quality—with user-configurable preference weights—would enable more nuanced routing policies tailored to specific application requirements.

Federated model registries. In multi-organization platforms, enabling controlled sharing of model configurations (e.g., sharing access to a fine-tuned model with partner organizations) requires a federated registry protocol with delegation and access control semantics that extend beyond the current four-scope hierarchy.

Semantic response caching. Deterministic or near-deterministic prompts (e.g., repeated classification tasks with identical inputs) could be served from a response cache, eliminating inference cost entirely for a subset of traffic. The challenge lies in defining cache key semantics for prompts that are semantically equivalent but textually distinct,

and in managing cache invalidation as models are updated.

The Model Proxy is deployed in production as part of the Kaman 3.0 enterprise AI platform, serving inference requests across multiple organizations with heterogeneous workloads. The architecture's stateless design, horizontal scalability, and zero-downtime extensibility make it suitable for deployment environments ranging from single-instance development setups to large-scale production clusters.

References

[1] BerriAI. LiteLLM: Call all LLM APIs using the OpenAI format. https://github.com/BerriAI/litellm, 2024.

[2] OpenRouter. OpenRouter: A unified interface for LLMs. https://openrouter.ai, 2024.

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

[4] 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.

[5] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[6] Martian. Model router: Intelligent LLM routing. Technical report, Martian, 2024.

[7] J. Wang, J. Li, and others. Mixture-of-Agents enhances large language model capabilities. *arXiv preprint arXiv:2406.04692*, 2024.

[8] OpenAI. Function calling and other API updates. Technical report, OpenAI, 2023.