vLLM Semantic Router: Signal Driven Decision Routing for Mixture-of-Modality Models

Xunzhuo Liu, Huamin Chen, Samzong Lu, Zhengda Tan, Senan Zedan,
Yehudit Kerido, Liav Weiss, Bishen Yu, Asaad Balum, Noa Limoy,
Abdallah Samara, Zhijie Wang, Qiping Pan, Rehan Khan, Zohaib Hassan[1]

mixdeers@gmail.com, hchen@redhat.com, samzong.lu@gmail.com, zdtna412@gmail.com, szedan@redhat.com, yehudit1987@gmail.com,
liavweiss@gmail.com, aukovyps@gmail.com, asaad.balum@gmail.com, noale7766@gmail.com, abdallahsamabd@gmail.com, i.zhijie.wang@outlook.com,
panqiping@outlook.com, rehan.irfan2002@gmail.com, zohaib179949@gmail.com

February 2026

Abstract

As large language models (LLMs) diversify across modalities, capabilities, and cost profiles, the problem of *intelligent request routing*—selecting the right model for each query at inference time—has become a critical systems challenge. We present **vLLM Semantic Router**, a signal-driven decision routing framework for Mixture-of-Modality (MoM) model deployments.

The central innovation is *composable signal orchestration*: the system extracts heterogeneous signal types from each request—from sub-millisecond heuristic features (keyword patterns, language detection, context length, role-based authorization) to neural classifiers (domain, embedding similarity, factual grounding, modality)—and composes them through configurable Boolean decision rules into deployment-specific routing policies. Different deployment scenarios—multi-cloud enterprise, privacy-regulated, cost-optimized, latency-sensitive—are expressed as different signal-decision configurations over the same architecture, without code changes.

Matched decisions drive *semantic model routing*: over a dozen of selection algorithms analyze request characteristics to find the best model cost-effectively, while per-decision plugin chains enforce privacy and safety constraints (jailbreak detection, PII filtering, hallucination detection via the three-stage *HaluGate* pipeline).

The system provides OpenAI API support for stateful multi-turn conversations, multi-endpoint and multi-provider routing across heterogeneous backends (vLLM, OpenAI, Anthropic, Azure, Bedrock, Gemini, Vertex AI), and a pluggable authorization factory supporting multiple auth providers. Deployed in production as an Envoy external processor, the architecture demonstrates that composable signal orchestration enables a single routing framework to serve diverse deployment scenarios with differentiated cost, privacy, and safety policies.

1 Introduction

The landscape of large language models has fragmented along multiple axes: modality (text, code, vision, diffusion), scale (1B to 1T+ parameters), cost (10× variation in per-token pricing), and specialization (general-purpose vs. domain-specific fine-tuning). Organizations increasingly

[1]Corresponding repository: https://github.com/vllm-project/semantic-router

operate *heterogeneous model fleets*—local vLLM instances alongside cloud endpoints from OpenAI, Anthropic, Azure, Bedrock, Gemini, and Vertex AI—each with different capabilities, pricing, and compliance characteristics. This heterogeneity creates a fundamental inference-time optimization problem: *given a user query, a fleet of diverse models, and deployment-specific constraints, which model should serve it, and what safety and privacy policies should apply?*

This problem is more nuanced than binary difficulty routing. A production routing system must simultaneously consider:

- **Multi-dimensional signals**: Query domain, modality, complexity, language, user identity, latency budgets, and real-time performance metrics all inform the optimal routing decision.
- **Privacy and safety**: Prompt injection, PII leakage, and hallucinated responses must be detected and mitigated—often with *different policies for different query types and user roles.*
- **Cost-effective model selection**: Algorithms must balance response quality against inference cost and latency, selecting from a heterogeneous pool of local and cloud-hosted models.
- **Deployment diversity**: The same routing framework must serve a privacy-regulated healthcare deployment (strict PII filtering, on-premise models only), a cost-optimized developer tool (aggressive caching, cheapest model first), and a multi-cloud enterprise (failover across providers)—through configuration, not code changes.
- **Multi-turn statefulness**: Routing decisions must be consistent across conversation turns, requiring stateful session management and context preservation.

Prior work on LLM routing has made significant progress on individual aspects. RouteLLM [26] trains classifiers to route between two models based on query difficulty. RouterDC [5] learns query-model embeddings via dual contrastive learning. AutoMix [2] formulates cascading as a POMDP. However, these approaches address model selection in isolation, without integrating signal extraction, safety enforcement, multi-provider backend management, or plugin extensibility into a unified framework.

1.1 Contributions

We present vLLM Semantic Router, a signal-driven decision routing system whose central innovation is **composable signal orchestration**: heterogeneous signals are extracted, composed through Boolean rules into deployment-specific decisions, and executed through per-decision plugin chains—enabling a single architecture to serve diverse deployment scenarios.

Our contributions are:

1. **Composable Signal-Decision-Plugin Architecture** (Sections 2 to 5): A three-layer architecture where eleven signal types are composed through Boolean decision rules into deployment-specific routing policies, with per-decision plugin chains for safety, caching, and augmentation. Different deployment scenarios (privacy-regulated, cost-optimized, multi-cloud) are expressed as different configurations over the same architecture.
2. **Semantic Model Routing with Cost-Aware Selection** (Section 9): A unified framework integrating thirteen model selection algorithms—rating-based, contrastive, cascading, classical ML, reinforcement learning, and latency-aware—that analyze request semantics to select the most cost-effective model while respecting per-decision privacy and safety constraints.
3. **HaluGate: Gated Hallucination Detection** (Section 7): A three-stage pipeline—sentinel gating, token-level detection, NLI-based explanation—that avoids unnecessary verification on non-factual queries while providing span-level diagnostics when hallucination is detected.

4. **Multi-Provider and Multi-Endpoint Routing** (Section 11): Native support for routing across heterogeneous backends (vLLM, OpenAI, Anthropic, Azure, Bedrock, Gemini, Vertex AI) with provider-specific protocol translation, a pluggable authorization factory for diverse auth mechanisms, weighted multi-endpoint load distribution, and full OpenAI Responses API support for stateful multi-turn conversations.

5. **LoRA-Based Multi-Task Classification** (Sections 8 and 10): A memory-efficient architecture using Low-Rank Adaptation that serves n classification tasks from a single base model with lightweight adapter heads, reducing aggregate model memory from n full copies to one base plus negligible adapter overhead.

1.2 Paper Organization

Section 2 presents the system architecture and composable orchestration model. Sections 3 and 4 formalize the signal extraction and decision evaluation layers. Sections 5 to 7 describe the plugin framework and safety subsystems. Sections 8 and 10 detail the LoRA-based classification architecture and multi-runtime inference design. Section 9 surveys the semantic model selection algorithms. Section 11 describes the multi-provider request processing pipeline. Sections 12 to 14 cover memory, observability, and deployment. Section 15 presents evaluation results. Section 16 discusses related work, and Section 17 concludes.

2 System Architecture

We formalize the routing problem and present the three-layer architecture that decomposes it into composable signal extraction, decision evaluation, and plugin execution—enabling a single framework to serve diverse deployment scenarios through configuration.

2.1 Problem Formulation

Let $\mathcal{M} = \{m_1, \ldots, m_K\}$ denote a set of K available model backends, each characterized by capability profile, cost, and latency. Each backend may be served by a different provider $p_k \in \mathcal{P}$ (e.g., local vLLM, OpenAI, Anthropic, Azure, Bedrock, Gemini), with provider-specific API protocols and authentication mechanisms. A deployment may expose multiple endpoints $\mathcal{E} = \{e_1, \ldots, e_L\}$ with weighted load distribution across backends.

Given an incoming request r (consisting of a message sequence, metadata, user identity, and headers), the routing problem is to:

1. Select a model $m^* \in \mathcal{M}$ that maximizes response quality while respecting cost and latency constraints;

2. Apply deployment-specific safety and privacy transformations $\mathcal{T}(r)$ before and after model invocation;

3. Route through the correct provider endpoint with appropriate authentication.

Naïve approaches either fix m^* statically or route based on a single dimension (e.g., estimated difficulty). We argue that production routing requires reasoning over *multiple orthogonal signal dimensions simultaneously*, with different *policies* (safety thresholds, caching strategies, prompt augmentation, model pools) for different routing outcomes—and that these policies must be *composable* to support diverse deployment scenarios without architectural changes.

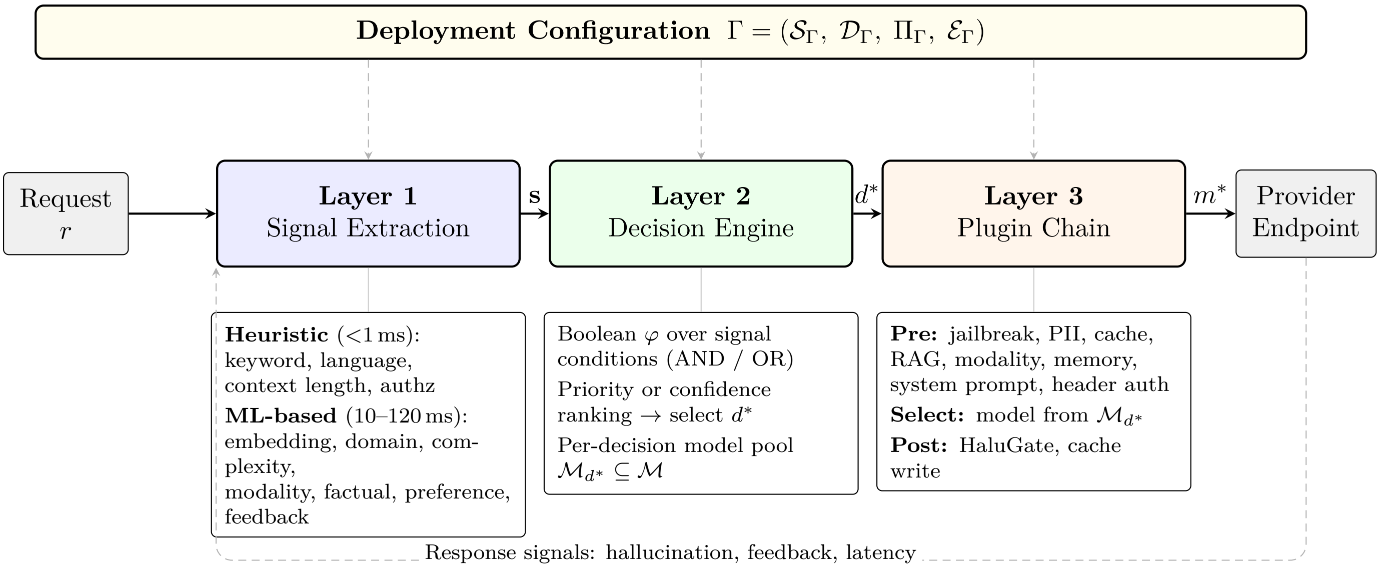

Figure 1: Three-layer architecture with closed-loop feedback. A deployment configuration Γ selects which signals, decisions, and plugins are active. Layer 1 extracts a signal vector $\mathbf{s}$ from the request. Layer 2 evaluates Boolean decision formulas to select d^*. Layer 3 executes the per-decision plugin chain, selects a model from d^*'s candidate set, and routes to the provider endpoint. Response-side signals feed back to enable adaptive routing.

2.2 Composable Signal Orchestration

The key architectural innovation is that the same signal extraction, decision evaluation, and plugin execution machinery can be *composed differently* for different deployment scenarios:

Definition 1 (Deployment Configuration)**.** *A deployment configuration* $\Gamma = (\mathcal{S}_\Gamma, \mathcal{D}_\Gamma, \Pi_\Gamma, \mathcal{E}_\Gamma)$ *specifies which signal types* $\mathcal{S}_\Gamma \subseteq \mathcal{S}$ *are active, what decisions* $\mathcal{D}_\Gamma$ *are evaluated, which plugin chains* Π_Γ *are attached, and which endpoints* $\mathcal{E}_\Gamma$ *are available.*

Example configurations:

- *Privacy-regulated (healthcare)*: Active signals include domain, authz, and language. Decisions route sensitive queries to on-premise models only. Plugins enforce strict PII redaction with no caching.
- *Cost-optimized (developer tool)*: Active signals include complexity, embedding, and keyword. Decisions cascade from cheap to expensive models. Plugins enable aggressive semantic caching.
- *Multi-cloud enterprise*: Active signals include domain, modality, and authz. Decisions distribute across multiple provider endpoints using latency-aware model selection with weighted failover. Plugins inject provider-specific auth headers.

All three scenarios use the same architecture; only Γ differs. This composability is the central design contribution.

2.3 Three-Layer Architecture

The architecture decomposes routing into three layers, each with a well-defined interface (Figure 1):

Layer 1: Signal Extraction. The signal layer maps a request r to a structured signal result $\mathbf{s}$, consisting of binary match indicators and real-valued confidences for each configured rule across eleven signal types. Heuristic signals (keyword, language, context length, authorization) complete in sub-millisecond time. ML-based signals (embedding similarity, domain classification, factual grounding, modality detection, complexity, preference, user feedback) require neural inference at 10–120 ms. Signals are evaluated in parallel, and only signal types referenced by at least one active decision are computed—a critical optimization for deployment configurations that use a subset of available signals.

Layer 2: Decision Evaluation. The decision layer takes the signal result $\mathbf{s}$ and evaluates a set of decisions $\mathcal{D} = \{d_1, \ldots, d_M\}$, each defined as a Boolean formula over signal conditions. The engine selects the best-matching decision d^* using either priority-based or confidence-weighted ranking. Each decision carries its own model candidate set $\mathcal{M}_{d^*} \subseteq \mathcal{M}$, enabling deployment-specific model pools (e.g., a privacy decision restricts candidates to on-premise models).

Layer 3: Plugin Chain. Each decision d^* carries a per-decision plugin configuration that defines: (a) *pre-routing plugins* (jailbreak detection, PII filtering, semantic caching, RAG context injection, modality routing, memory retrieval, system prompt augmentation, header mutation for provider auth), executed before model invocation; (b) a *semantic model selection algorithm* applied to d^*'s candidate model set $\mathcal{M}_{d^*}$ to find the best model cost-effectively; (c) *post-routing plugins* (hallucination detection, cache updates), executed on the model response.

2.4 Design Principles

Four principles guide the architecture:

Composability. Complex routing policies are expressed as compositions of simple primitives: Boolean combinations of signal conditions form decisions; sequences of typed plugins form execution chains; deployment scenarios are expressed as configuration profiles. This avoids monolithic routing logic and enables the same system to serve fundamentally different deployment requirements.

Orthogonality. Signals, decisions, and plugins are independent modules with a uniform interface boundary. New signal types can be added by implementing a single evaluation function—the decision engine references signals solely by type and rule name, requiring no modification. Likewise, new plugins and providers are registered independently. The current eleven signal types are the built-in set; the framework is designed to be extended with domain-specific signals as deployment requirements evolve.

Closed-loop adaptivity. The bidirectional signal flow described in Section 3 enables the architecture to operate as a *closed-loop control system* [1]. In control-theoretic terms, the signal–decision–plugin pipeline is the *plant*, response-side signals (hallucination detection, user feedback, latency measurements) are the *sensors*, and a policy adjustment mechanism is the *controller* that updates decision parameters $\theta^{(t)}$ (priorities, model weights) based on observed response quality:

$$\theta^{(t+1)} = \theta^{(t)} + \eta \, \nabla_\theta \, \mathbb{E}\big[Q\big(r,\, m^*(r; \theta^{(t)})\big)\big] \tag{1}$$

where $Q(r, m^*)$ is a response quality metric and η is a learning rate. This formulation connects to the *contextual bandit* framework [19]: the signal vector $S(r)$ serves as the context, model selection is the action, and response quality is the reward. Standard regret bounds from online learning theory [31] guarantee that the cumulative routing quality of such an adaptive policy converges to that of the best fixed policy in hindsight at a rate of $O(\sqrt{T})$, providing formal performance guarantees for self-improving routing.

Per-decision scoping. Safety thresholds, caching policies, model candidates, and auth mechanisms are scoped to individual decisions rather than applied globally. A coding-focused decision can disable PII detection while a customer-support decision enforces strict filtering—using the same system configuration.

Provider abstraction. The architecture abstracts over provider-specific protocols, authentication, and endpoint topologies. Multi-endpoint routing with weighted distribution and failover is handled at the infrastructure layer, enabling decisions to reference models by capability rather than by provider-specific endpoint.

3 Signal Extraction Layer

The signal extraction layer maps an incoming request r to a structured signal result that characterizes the request along eleven orthogonal dimensions. We formalize the signal model and describe the extraction algorithms.

3.1 Signal Model

Definition 2 (Signal Rule)**.** *A* signal rule $\rho = (\tau, n, f)$ *consists of a signal type* $\tau \in \mathcal{T}$*, a rule name* n*, and an evaluation function* $f : \mathcal{R} \to \{0,1\} \times [0,1]$ *that maps a request to a binary match indicator and a confidence score.*

Definition 3 (Signal Result)**.** *Given a rule set* $\mathcal{R} = \{\rho_1, \ldots, \rho_N\}$*, the signal result for request* r *is:*

$$S(r) = \{(\rho_i,\ \mathbf{1}[f_i(r)],\ c_i(r)) \mid \rho_i \in \mathcal{R}\} \tag{2}$$

where $\mathbf{1}[f_i(r)]$ *is the match indicator and* $c_i(r) \in [0,1]$ *is the confidence.*

The eleven signal types partition into *heuristic* and *learned* categories based on whether they require neural inference.

3.2 Heuristic Signals

Heuristic signals use deterministic or statistical algorithms with sub-millisecond latency:

Keyword (τ_{kw}). Rules are defined as pattern sets with Boolean combinators. Each rule specifies a set of patterns $P = \{p_1, \ldots, p_k\}$ with a combinator $\in \{\text{AND}, \text{OR}, \text{NOR}\}$ and one of three matching methods:

- *Regex* (default): compiled regular expressions with word boundaries; confidence is 1.0 on match.
- *BM25*: BM25 scoring dispatched to a Rust-backed classifier via FFI. Each keyword is scored against the request using TF-IDF term weighting, and the rule matches when the score exceeds a configurable threshold (default 0.1). Confidence is derived from the BM25 score, providing a graded relevance signal rather than a binary match.
- *N-gram*: character n-gram similarity (default trigram) dispatched to the same Rust binding. The rule matches when the Jaccard similarity between the keyword and request n-gram sets exceeds a threshold (default 0.4), providing inherent tolerance to typos and morphological variation without a dedicated fuzzy-matching pass.

For AND: $f(r) = \bigwedge_i \text{match}(p_i, r)$; for OR: $f(r) = \bigvee_i \text{match}(p_i, r)$. The combinators apply uniformly across all three methods.

Context Length (τ_{ctx}). Rules define token-count intervals $[l, u]$. Given estimated token count $t(r)$, the rule matches iff $l \leq t(r) \leq u$. This enables complexity-aware routing (e.g., short queries to fast models, long contexts to extended-context models).

Language (τ_{lang}). Rules bind detected language codes to named signals using statistical n-gram detection over 100+ languages. Enables language-specific routing (e.g., CJK queries to multilingual-specialized models).

Authorization (τ_{authz}). Role-based access control signals extracted from request headers, supporting a pluggable authentication factory. The authz signal layer abstracts over multiple identity providers (API key, OAuth2/OIDC, cloud IAM, custom JWT, LDAP) through provider-specific extractors that resolve user identities and group memberships from credentials. Role bindings then map resolved identities to named signals, enabling per-role routing policies (e.g., premium users routed to higher-quality models, free-tier users restricted to cost-effective models). This *inbound* authorization (who is the user and what can they access?) is complementary to the *outbound* authorization factory (Section 11.5) that injects provider-specific credentials when forwarding to backends.

3.3 Learned Signals

Learned signals require neural inference, typically 10–100 ms, using the LoRA-based classifiers described in Section 8:

Embedding Similarity (τ_{emb}). Each rule defines reference texts $\{t_1, \ldots, t_k\}$ and a similarity threshold θ. The request embedding $\mathbf{e}_r$ is computed via a shared embedding model, and the rule matches iff:

$$\max_i \cos(\mathbf{e}_r, \mathbf{e}_{t_i}) \geq \theta \tag{3}$$

The confidence equals the maximum cosine similarity. This provides scalable semantic matching without per-rule model training.

Domain Classification (τ_{dom}). A LoRA-adapted classifier trained on MMLU categories maps requests to domain labels (STEM, humanities, code, creative writing, etc.). The classification confidence serves as the signal confidence.

Factual Grounding (τ_{fact}). A binary classifier (the HaluGate Sentinel, Section 7) determines whether the query requires factual verification, distinguishing factual questions from creative or code-generation tasks.

User Feedback (τ_{fb}). A multi-class classifier detects satisfaction, dissatisfaction, clarification requests, and preference for alternatives, enabling feedback-driven routing adjustments.

Modality (τ_{mod}). A three-class classifier (autoregressive, diffusion, both) determines the appropriate model modality for the request, trained on mixed text-generation and image-generation datasets.

Complexity (τ_{cpx}) and **Preference** (τ_{pref}). Additional learned signals for query difficulty estimation and personalized routing based on user interaction history.

3.4 Parallel Evaluation with Lazy Computation

A key optimization is *demand-driven evaluation*: the engine computes only signal types referenced by at least one configured decision. Let $\mathcal{T}_{\text{used}} = \bigcup_{d \in \mathcal{D}} \{\tau \mid \exists \, \text{condition in } d \text{ of type } \tau\}$. Signal evaluators for types in $\mathcal{T}_{\text{used}}$ are launched as concurrent coroutines, with heuristic signals completing before learned signals due to their sub-millisecond latency.

This demand-driven approach avoids the cost of unused signal types. In typical configurations with 3–5 active signal types out of eleven, this reduces total signal extraction latency by 50–70% compared to exhaustive evaluation.

3.5 Extensibility

The eleven signal types described above represent the current built-in set; the framework is not limited to these. The signal extraction layer defines a uniform interface—each signal type implements an evaluation function $f : \mathcal{R} \to \{0, 1\} \times [0, 1]$—and the decision engine references signals solely by type and rule name. Adding a new signal type requires only implementing this interface and registering the type; no changes to the decision engine, plugin chain, or deployment infrastructure are needed. This open architecture allows operators to introduce domain-specific signals (e.g., regulatory compliance classifiers, custom toxicity detectors) alongside the built-in types.

3.6 Bidirectional Signal Flow

Signals are not limited to the inbound request path. The system also extracts signals from model *responses*, enabling closed-loop routing policies that adapt based on output characteristics. The primary example is HaluGate (Section 7): the Sentinel classifier on the request path determines whether a query requires factual verification (the τ_{fact} signal), and if so, the Detector and Explainer stages analyze the model's response for unsupported claims—producing response-side detection results (confidence scores, hallucinated spans, NLI explanations) that are propagated via HTTP headers or body annotations. This bidirectional flow—request signals gating which response analyses to perform, and response signals feeding back into observability and policy enforcement—enables adaptive quality assurance without imposing uniform overhead on all requests.

3.7 Information-Theoretic Signal Analysis

With N signal types evaluated per request, a natural question is whether all signals contribute independently to routing quality or whether some carry redundant information. Information theory provides the formal framework for this analysis [32].

For a signal type τ_i and the routing outcome variable Y (the selected model), the *mutual information* $I(\tau_i; Y)$ quantifies the reduction in uncertainty about the routing decision provided by observing signal τ_i. The *conditional mutual information* $I(\tau_i; Y \mid \tau_j)$ measures the additional information from τ_i given that τ_j is already observed. When $I(\tau_i; Y \mid \tau_j) \approx 0$, signals τ_i and τ_j are redundant with respect to routing—observing both provides no more discriminative power than observing one.

This analysis enables two optimizations. First, *adaptive signal pruning*: in a given deployment configuration, signals with near-zero mutual information with the routing outcome can be disabled without affecting routing quality, reducing extraction latency beyond the demand-driven approach of Section 3. Second, *information-ordered evaluation*: evaluating high-$I(\tau_i; Y)$ signals first and short-circuiting when the decision outcome is already determined—analogous to early termination in decision trees—can reduce average per-request evaluation cost. The minimum description length (MDL) principle [29] provides a complementary perspective: the optimal signal subset is the one that describes the routing policy with minimum total code length, balancing signal extraction cost against routing precision.

4 Decision Engine

The decision engine evaluates a set of routing decisions against the signal result and selects the best match. We formalize the decision model, present the evaluation algorithm, and analyze the selection strategies.

4.1 Decision Model

Definition 4 (Decision)**.** *A* decision $d = (n, \phi, \mathcal{M}_d, \Pi_d, p)$ *consists of a name* n*, a Boolean formula* ϕ *over signal conditions, a candidate model set* $\mathcal{M}_d \subseteq \mathcal{M}$*, a plugin configuration* Π_d*, and a priority* $p \in \mathbb{Z}$*.*

Definition 5 (Signal Condition)**.** *A* signal condition $\gamma = (\tau, n, \nu)$ *references a signal type* τ*, a rule name* n*, and an optional negation flag* $\nu \in \{0, 1\}$*. The base satisfaction is:*

$$sat_0(\gamma, S(r)) = \mathbf{1}\left[\exists\, (\rho, 1, c) \in S(r) : \rho.\tau = \tau \wedge \rho.n = n\right] \tag{4}$$

With negation applied:

$$sat(\gamma, S(r)) = \begin{cases} 1 - sat_0(\gamma, S(r)) & \textit{if } \nu = 1 \\ sat_0(\gamma, S(r)) & \textit{otherwise} \end{cases} \tag{5}$$

Negation enables exclusion patterns: a condition $(\tau_{\text{dom}}, \text{“code”}, 1)$ matches requests that are *not* classified as code, allowing decisions to route non-code queries without enumerating all other domains.

Definition 6 (Rule Formula)**.** *A* rule formula $\phi = (\textsc{op}, \Gamma)$ *combines conditions* $\Gamma = \{\gamma_1, \ldots, \gamma_L\}$ *with an operator* $\textsc{op} \in \{\textit{AND}, \textit{OR}\}$*:*

$$eval(\phi, S(r)) = \begin{cases} \bigwedge_{j=1}^{L} sat(\gamma_j, S(r)) & \textit{if } \textsc{op} = \textit{AND} \\ \bigvee_{j=1}^{L} sat(\gamma_j, S(r)) & \textit{if } \textsc{op} = \textit{OR} \end{cases} \tag{6}$$

The combination of per-condition negation with AND/OR operators and inter-decision priority yields full Boolean expressiveness: AND with negated conditions expresses “match A but not B”; OR with negated conditions expresses “match unless both A and B”; priority-ordered decisions with progressively broader conditions implement fallback chains. This model is deliberately kept flat rather than nested: it provides interpretable, auditable routing policies that administrators can reason about directly, while achieving complex routing logic through composition of multiple prioritized decisions.

4.2 Confidence Computation

When a decision matches, we compute a confidence score as the mean confidence over satisfied conditions:

$$\text{conf}(d, S(r)) = \frac{1}{|\Gamma_{\text{sat}}|} \sum_{\gamma_j \in \Gamma_{\text{sat}}} c_j(r) \tag{7}$$

where $\Gamma_{\text{sat}} = \{\gamma_j \in \Gamma \mid \text{sat}(\gamma_j, S(r)) = 1\}$ and $c_j(r)$ is the signal confidence for condition γ_j. For embedding signals, c_j is the cosine similarity; for heuristic and binary ML signals, $c_j = 1.0$.

4.3 Selection Strategies

Given the set of matched decisions $\mathcal{D}_{\text{match}} = \{d \in \mathcal{D} \mid \text{eval}(\phi_d, S(r)) = 1\}$, two strategies select d^*:

Priority Strategy.

$$d^* = \arg\max_{d \in \mathcal{D}_{\text{match}}} p_d \tag{8}$$

This provides deterministic, administrator-controlled routing. Ties are broken by insertion order.

Confidence Strategy.

$$d^* = \arg\max_{d \in \mathcal{D}_{\text{match}}} \text{conf}(d, S(r)) \tag{9}$$

This enables data-driven routing where embedding similarity and classifier confidence drive selection.

The priority strategy is the default for production deployments where predictability is paramount. The confidence strategy is preferred for experimental settings where the system should adapt to query characteristics.

Algorithm 1 Decision Evaluation

Require: Signal result $S(r)$, decisions $\mathcal{D}$, strategy $\sigma \in \{\text{priority}, \text{confidence}\}$
Ensure: Selected decision d^*, confidence c^*

$\mathcal{D}_{\text{match}} \leftarrow \emptyset$
for $d \in \mathcal{D}$ **do**
  if $\text{eval}(\phi_d, S(r))$ **then**
    $c_d \leftarrow \text{conf}(d, S(r))$
    $\mathcal{D}_{\text{match}} \leftarrow \mathcal{D}_{\text{match}} \cup \{(d, c_d)\}$
  end if
end for
if $\sigma = \text{priority}$ **then**
  $(d^*, c^*) \leftarrow \arg\max_{(d,c)\in\mathcal{D}_{\text{match}}} p_d$
else
  $(d^*, c^*) \leftarrow \arg\max_{(d,c)\in\mathcal{D}_{\text{match}}} c$
end if
return (d^*, c^*)

4.4 Evaluation Algorithm

The algorithm runs in $O(M \cdot L_{\max})$ where $M = |\mathcal{D}|$ is the number of decisions and $L_{\max}$ is the maximum number of conditions per decision. In practice, $M \leq 50$ and $L_{\max} \leq 10$, making decision evaluation negligible ($< 0.1\,\text{ms}$) relative to signal extraction.

4.5 Expressiveness Analysis

The Boolean combination model can express common routing patterns:

- **Domain routing**: A single domain condition (OR with one condition) routes by classified domain.
- **Guarded routing**: AND of a domain condition and a complexity condition routes complex queries within a domain to a capable model.
- **Exclusion routing**: AND of a domain condition and a *negated* complexity condition routes simple queries within a domain to a lightweight model, avoiding the cost of a full-capability model for straightforward requests.
- **Multi-signal routing**: AND of keyword, embedding, and language conditions provides precise routing for specific query patterns.
- **Fallback chains**: Multiple decisions with decreasing priority and progressively broader conditions implement fallback routing.

Functional completeness. The expressiveness of this model rests on a classical result from Boolean algebra [12]: the operator set $\{\wedge, \vee, \neg\}$ is *functionally complete*—any Boolean function $f : \{0,1\}^N \to \{0,1\}$ can be expressed as a formula over these operators. In our setting, the N signal match indicators form the input bits, and each decision formula ϕ computes a Boolean function over them using AND, OR, and per-condition NOT. A single flat formula can express any clause (conjunction or disjunction of literals); priority-ordered decisions then compose multiple clauses into an ordered evaluation, equivalent to a decision list [30] over Boolean features.

Proposition 1. *For any routing policy expressible as a function* $\pi : \{0,1\}^N \to \mathcal{M} \cup \{\perp\}$ *mapping signal vectors to model selections, there exists a decision set* $\mathcal{D}$ *with AND/OR/NOT formulas and priority ordering such that* π *is realized by the evaluation algorithm (Algorithm 1).*

Proof sketch. Express π in disjunctive normal form: for each model m_k, collect the minterms (signal vectors) that map to m_k and form their disjunction. Each disjunction becomes a decision d_k with $\textsc{op} = \textsc{or}$ over AND-clauses (each AND-clause is itself a decision at higher priority with $\textsc{op} = \textsc{and}$ over literals). Since $\{\wedge, \vee, \neg\}$ is functionally complete, every minterm is expressible, and priority ordering resolves overlaps deterministically. □

This universality guarantee means the decision engine imposes *no inherent limitation* on what routing policies can be configured—any policy that depends on the binary signal outcomes is realizable.

Structural analogy to programmable logic. The signal–decision architecture is structurally isomorphic to a *Programmable Logic Array* (PLA) [8], a well-studied circuit primitive that implements arbitrary two-level Boolean functions:

PLA Component	**Routing System**	**Role**
Input lines	Signal extractors ($\tau_1, \ldots, \tau_N$)	Produce binary feature vector
AND-plane (product terms)	Decision AND-formulas	Compute conjunctions of literals
OR-plane (sum terms)	Decision OR-formulas	Compute disjunctions of clauses
Priority encoder	Priority-ordered evaluation	Select highest-priority match
Output lines	Model selection + plugin chain	Execute the routed action

In a PLA, the AND-plane and OR-plane are "programmed" by setting fuse connections; in our system, they are programmed by YAML configuration. Just as a PLA can implement any Boolean function within its capacity (number of product terms and inputs), the decision engine can implement any routing policy within its configured signal and decision budget. This correspondence provides a well-understood theoretical foundation: decades of results on PLA minimization, hazard-free design, and testability [4] apply directly to reasoning about decision set optimization and coverage analysis.

Decision set verification and minimization. The PLA correspondence makes formal analysis tools from logic synthesis directly applicable to routing configurations. *Coverage analysis* checks whether every reachable point in the signal space $\{0,1\}^N$ is matched by at least one decision, identifying dead zones where requests would receive no routing directive. *Conflict detection* identifies signal combinations where multiple decisions match but route to incompatible model pools, flagging ambiguities that priority ordering must resolve. *Decision minimization*, analogous to the Espresso heuristic for two-level logic optimization [4], can reduce a decision set to a minimal equivalent form by merging decisions with compatible conditions and eliminating subsumed rules. These standard logic-verification techniques become applicable to routing policy validation without adaptation, a direct consequence of the structural isomorphism.

4.6 Generalization to Fuzzy Evaluation

The Boolean decision model admits a natural generalization when signal confidence scores are continuous. Rather than binarizing each signal's output before Boolean combination, we evaluate decision formulas over the continuous confidence values directly, using fuzzy logic operators [38].

Definition 7 (Fuzzy Rule Evaluation)**.** *Given continuous signal confidences $c_j(r) \in [0,1]$ for each condition γ_j, and letting $\tilde{c}_j(r) = 1 - c_j(r)$ when γ_j is negated and $\tilde{c}_j(r) = c_j(r)$ otherwise, the fuzzy evaluation of a rule formula $\phi = (\textsc{op}, \Gamma)$ is:*

$$\widetilde{eval}(\phi, S(r)) = \begin{cases} \min_{j=1}^{L} \tilde{c}_j(r) & \text{if } \textsc{op} = AND \\ \max_{j=1}^{L} \tilde{c}_j(r) & \text{if } \textsc{op} = OR \end{cases} \tag{10}$$

The operators $(\min, \max, 1-x)$ form the standard fuzzy complement triple and satisfy De Morgan's laws, preserving the algebraic properties of the crisp model [3]. This fuzzy evaluation is a *strict generalization*: when all confidences are binary ($c_j \in \{0,1\}$), min reduces to $\wedge$ and max reduces to $\vee$, so the evaluation coincides exactly with the crisp Boolean model.

The practical consequence is significant. The current confidence strategy (Equation (7)) uses mean confidence as a tiebreaker *after* binary matching. Fuzzy evaluation incorporates confidence *during* formula evaluation: a decision with three conditions matched at confidences $(0.95, 0.88, 0.72)$ yields a fuzzy AND score of 0.72, while a decision with two conditions at $(0.99, 0.98)$ scores 0.98—correctly preferring the more confident partial match even when both decisions pass binary evaluation. The functional completeness result of the previous section extends directly: the fuzzy operator triple $(\min, \max, 1-x)$ is functionally complete over the continuous lattice $[0,1]$, so any monotone routing policy over continuous signal scores is realizable.

4.7 Composable Decision Profiles

The decision model directly enables *composable signal orchestration*: different deployment scenarios are expressed as different decision sets $\mathcal{D}$ over the same signal infrastructure. A healthcare deployment defines decisions with authz and domain conditions routing to compliant model pools; a developer-tool deployment defines decisions with complexity and keyword conditions routing to cost-optimized cascades; a multi-cloud deployment defines decisions with domain and modality conditions, using latency-aware model selection across provider endpoints.

Formally, switching deployment scenarios corresponds to loading a different decision profile $\mathcal{D}_\Gamma$, while the signal extraction layer $\mathcal{S}$ and plugin implementations Π remain unchanged. This separation of *policy* (what decisions to evaluate) from *mechanism* (how signals are computed and plugins execute) is the architectural basis for the composability claimed in Section 2.

5 Plugin Framework

The plugin layer provides a composable middleware architecture where each matched decision activates an independent chain of typed transformations. We describe the plugin model and three core infrastructure plugins; safety plugins are covered in Sections 6 and 7, and memory retrieval and RAG injection are covered in Section 12.

5.1 Plugin Execution Model

Formally, a plugin π is a typed transformation on the request-response pair:

$$\pi : (\text{Request}, \text{Context}, \text{Config}_\pi) \to (\text{Request}', \text{Response}') \cup \{\bot\} \tag{11}$$

where $\bot$ denotes early termination (e.g., a cache hit returning immediately or a safety violation blocking the request).

Plugins execute in a fixed pipeline order within each decision's chain. On the *request path*: jailbreak $\to$ PII $\to$ cache $\to$ RAG $\to$ modality $\to$ memory $\to$ system prompt $\to$ header mutation.

On the *response path*: hallucination detection → cache write. Each plugin is independently enabled per decision, and its configuration (thresholds, modes, policies) is scoped to that decision.

This per-decision scoping is a key architectural distinction from systems that apply safety and caching globally: it allows differentiated policies for different routing outcomes within the same deployment.

5.2 Semantic Cache

The semantic cache exploits the observation that semantically similar queries often produce equivalent responses, avoiding redundant model invocations.

Similarity model. Given a query q extracted from request r, the cache searches for an entry e such that:

$$\cos(\mathbf{e}_q, \mathbf{e}_e) \geq \theta_d \tag{12}$$

where $\mathbf{e}_q, \mathbf{e}_e$ are embeddings computed by the shared embedding model and θ_d is the per-decision similarity threshold. On hit, the cached response is returned immediately, bypassing model invocation entirely.

Write-through protocol. On cache miss, a pending entry is registered before forwarding to the model. Upon receiving the response, the entry is completed with the response content. This ensures that concurrent identical queries observe the pending state rather than triggering redundant model calls.

Backend abstraction. Four backends provide different latency-persistence tradeoffs: (1) in-memory HNSW for single-node low-latency deployments; (2) Redis for distributed persistent caching; (3) Milvus for large-scale approximate nearest neighbor search; (4) a hybrid two-tier design combining in-memory HNSW (fast path) with Milvus (persistent store).

5.3 System Prompt Injection

Per-decision system prompt injection enables different routing paths to carry different instructions. Two composition modes are defined:

- **Replace**: Substitutes the entire system message, providing complete control over the model's behavioral context.
- **Insert**: Prepends the decision's prompt to the existing system message, augmenting without overriding user-provided instructions.

This enables patterns such as injecting domain-specific instructions for expert routing or safety preambles for sensitive query categories.

5.4 Header Mutation

Header mutation enables metadata propagation to downstream model backends via HTTP header modifications (add, update, delete). This supports use cases including: backend-specific authentication injection, routing decision metadata propagation for downstream observability, and custom signaling to model-serving frameworks (e.g., LoRA adapter selection via headers).

6 Request-Time Safety: Jailbreak and PII Detection

Request-time safety plugins detect adversarial inputs and sensitive information before queries reach model backends. Both operate as gating plugins that can terminate request processing with an immediate rejection.

6.1 Jailbreak Detection

Jailbreak attacks [14] attempt to override model safety instructions through adversarial prompt construction. Our detection pipeline addresses this as a binary classification problem with per-decision sensitivity control.

Formulation. Given a text input x (the user's latest message, or the full conversation history when context-aware detection is enabled), a classifier g_{jb} produces:

$$g_{\text{jb}}(x) = (y, c) \in \{\text{BENIGN}, \text{INJECTION}, \text{JAILBREAK}\} \times [0, 1] \tag{13}$$

The request is blocked iff $y \neq \text{BENIGN}$ and $c \geq \theta_d$, where θ_d is the per-decision threshold.

Classifier architecture. We support four inference backends with varying context-length and deployment characteristics: (1) BERT with LoRA adapters (standard context); (2) ModernBERT [35] with Flash Attention; (3) mmBERT-32K with YaRN RoPE for 32K-token contexts; (4) external vLLM-served guardrail models for decoupled scaling. All local backends use the LoRA adapter architecture (Section 8), reducing model memory footprint.

Per-decision sensitivity. Different decisions configure different thresholds: a public-facing endpoint might use $\theta = 0.7$ for aggressive detection, while an internal developer tool uses $\theta = 0.95$ to minimize false positives. History inclusion is also per-decision: some decisions analyze only the latest message (low latency), while others analyze the full conversation (higher recall for multi-turn attacks).

6.2 PII Detection

PII detection identifies personally identifiable information at the token level and enforces configurable allow/deny policies.

Formulation. A token classifier g_{pii} operates on the input sequence $x = (x_1, \ldots, x_T)$ and produces BIO-tagged predictions:

$$g_{\text{pii}}(x) = \{(i, j, \ell, c) \mid \text{span } x_i \ldots x_j \text{ is PII type } \ell \text{ with confidence } c\} \tag{14}$$

where $\ell \in \{\text{PERSON}, \text{EMAIL}, \text{PHONE}, \text{SSN}, \text{CREDIT\_CARD}, \ldots\}$.

Policy model. Detected PII entities are evaluated against a per-decision policy:

$$\text{allowed}(\ell) = \begin{cases} \ell \notin \mathcal{L}_{\text{deny}} & \text{if allow-by-default} \\ \ell \in \mathcal{L}_{\text{allow}} & \text{otherwise} \end{cases} \tag{15}$$

If any entity $(\cdot, \cdot, \ell, c)$ satisfies $c \geq \theta_d$ and $\neg\text{allowed}(\ell)$, the request is blocked.

This two-mode policy (allow-by-default with deny list, or deny-by-default with allow list) provides flexible control: a medical application might allow PERSON while blocking SSN, whereas a general chatbot might block all PII types.

6.3 Safety Pipeline Ordering

Jailbreak detection executes before PII detection, ensuring that adversarial inputs designed to bypass PII detection are caught first. Both use the LoRA adapter architecture (Section 8), and when run concurrently via parallel goroutines, the wall-clock time is dominated by the slower of the two classifiers rather than their sum.

7 HaluGate: Gated Hallucination Detection

Hallucination—generating plausible but unsupported content—is a fundamental limitation of autoregressive language models [22, 25]. We introduce HaluGate, a three-stage pipeline that addresses a key efficiency challenge: most queries (creative writing, code generation, brainstorming) do not require factual verification, yet naïve hallucination detection incurs overhead on every response.

7.1 Design Rationale

Existing approaches apply hallucination detection uniformly to all responses [22] or require multiple response samples [25]. HaluGate introduces two innovations: (1) a *gating stage* that skips verification for non-factual queries, amortizing detection cost over the query distribution; and (2) a *span-level* detection and explanation pipeline that identifies *which* tokens are hallucinated and *why*, rather than providing only a binary judgment.

7.2 Three-Stage Pipeline

Stage 1: Sentinel (Gating). A lightweight binary classifier g_{sent} determines whether the query warrants factual verification:

$$g_{\text{sent}}(q) \in \{\text{NEEDS\_FACT\_CHECK}, \text{NO\_FACT\_CHECK}\} \tag{16}$$

If $g_{\text{sent}}(q) = \text{NO\_FACT\_CHECK}$, Stages 2–3 are skipped entirely. The Sentinel operates on the request text and is implemented as a LoRA-adapted classifier sharing the base model with other signal extractors. In practice, 40–60% of queries are classified as non-factual, proportionally reducing the average detection cost.

The Sentinel also serves dual duty as the `fact_check` signal in the signal extraction layer (Section 3), enabling decisions to incorporate factual grounding into routing logic.

Stage 2: Detector (Span Identification). A token-level classifier g_{det} identifies hallucinated spans in the model response:

$$g_{\text{det}}(q, \mathbf{c}, a) = \{(i, j, c_{ij}) \mid a_i \dots a_j \text{ is unsupported by context } \mathbf{c}\} \tag{17}$$

where q is the user query, $\mathbf{c}$ is the grounding context (user-provided context and tool-call results), a is the assistant's response, and (i, j, c_{ij}) denotes a flagged span with confidence.

When tool-calling is present, tool execution results provide high-quality ground truth: database query results, API responses, and calculations serve as authoritative context $\mathbf{c}$, substantially improving detection precision.

Stage 3: Explainer (NLI Classification). For each flagged span (i, j), a Natural Language Inference (NLI) model [36] classifies the relationship between the span and the grounding context:

$$g_{\text{nli}}(a_{i:j}, \mathbf{c}) \in \{\text{ENTAILMENT}, \text{CONTRADICTION}, \text{NEUTRAL}\} \tag{18}$$

This distinguishes between content that *contradicts* the context (definitive hallucination) and content that is merely *unsupported* (potential hallucination), providing actionable diagnostics.

7.3 Cost Analysis

Let p_{factual} be the fraction of queries requiring factual verification, C_{sent}, C_{det}, C_{nli} be the costs of each stage, and $\bar{k}$ be the average number of flagged spans. The expected cost per query is:

$$\mathbb{E}[\text{Cost}] = C_{\text{sent}} + p_{\text{factual}} \cdot (C_{\text{det}} + \bar{k} \cdot C_{\text{nli}}) \tag{19}$$

Since the Sentinel is a lightweight LoRA-adapted classifier (Section 8) that runs concurrently with other signal extractors, its wall-clock cost is largely hidden behind other ML signals. For a workload with $p_{\text{factual}} = 0.5$, the gating stage reduces the expected Detector and Explainer cost by approximately 50% compared to applying full detection to all responses.

7.4 Action Policies

HaluGate supports four configurable response actions:

The progressive architecture enables incremental deployment: organizations begin with Sentinel-only gating (signal-layer integration at minimal cost), add the Detector for span-level monitoring, and enable the Explainer for full diagnostic output.

Table 1: HaluGate action policies upon hallucination detection

Action	Semantics
`block`	Reject the response; return an error to the client. Appropriate for high-stakes factual applications.
`header`	Propagate detection metadata via HTTP headers, enabling downstream policy enforcement by the client or API gateway.
`body`	Prepend a warning to the response body, alerting users to potential inaccuracies.
`none`	Log detection results without modifying the response. Useful for monitoring and threshold calibration.

8 LoRA-Based Multi-Task Classification and MoM Model Family

Signal-driven routing requires multiple classification tasks on the critical path of every request. Naïvely, each task requires a separate fine-tuned model, creating a memory scaling problem. We describe the LoRA-based architecture that addresses this and the purpose-built model family trained for semantic routing.

8.1 Problem: Linear Memory Scaling

Let n denote the number of active classification tasks (domain, jailbreak, PII, fact-check, feedback, modality). With independently fine-tuned models, the total model memory is:

$$M_{\text{indep}} = n \cdot |\theta_{\text{base}}| \tag{20}$$

where $|\theta_{\text{base}}|$ is the parameter count of a single base model. For $n = 6$ tasks with a 150M-parameter base model, this requires storing and loading six full model copies ($\sim$900M parameters total)—a significant memory burden, especially in GPU-constrained environments.

Additionally, managing n independent model checkpoints complicates deployment, versioning, and updates.

8.2 Solution: Single Base Model with LoRA Adapters

Low-Rank Adaptation (LoRA) [9] represents task-specific weight modifications as low-rank decompositions:

$$W_i' = W + \Delta W_i = W + B_i A_i, \quad B_i \in \mathbb{R}^{d\times r},\ A_i \in \mathbb{R}^{r\times d} \tag{21}$$

where W is the shared base weight, $r \ll d$ is the adapter rank, and $B_i A_i$ is the task-specific perturbation.

The aggregate model memory becomes:

$$M_{\text{LoRA}} = |\theta_{\text{base}}| + \sum_{i=1}^{n} 2rd = |\theta_{\text{base}}| + n \cdot 2rd \tag{22}$$

With rank $r = 32$ and hidden dimension $d = 768$, each adapter adds $2 \times 32 \times 768 =$ 49,152 parameters ($\sim$0.02% of the base model). For $n = 6$, total adapter overhead is $\sim$295K parameters—negligible compared to the 150M-parameter base.

Memory reduction.

$$\frac{M_{\text{LoRA}}}{M_{\text{indep}}} = \frac{|\theta_{\text{base}}| + n \cdot 2rd}{n \cdot |\theta_{\text{base}}|} \approx \frac{1}{n} \quad \text{for } 2nrd \ll |\theta_{\text{base}}| \tag{23}$$

At $n = 6$, this yields $\sim 6\times$ memory reduction: one 150M-parameter model plus six tiny adapters instead of six full copies.

8.3 Inference Architecture

Each classification task proceeds as a full forward pass through the base model with the task-specific LoRA perturbation applied (Figure 2):

1. **Load**: A single base model is loaded into GPU/CPU memory at startup. Each LoRA adapter (a pair of small matrices per adapted layer) is loaded alongside it.
2. **Inference**: For each classification task i, the base model runs a forward pass with adapter i's weights merged: $W_i' = W + B_i A_i$. Each task still requires its own forward pass.
3. **Parallelism**: Multiple classification tasks execute concurrently via parallel threads/goroutines. Wall-clock time is determined by the slowest classifier, not the sum.

Note that LoRA does *not* eliminate the per-task forward pass—each adapter requires a full inference through the modified model. The primary benefit is **memory efficiency**: deploying six classifiers requires the memory footprint of approximately one model rather than six, and all adapters can be updated independently without reloading the base model.

8.4 MoM Model Family

We train a family of purpose-built models (MoM: Mixture-of-Models) optimized for routing classification tasks:

Table 2: MoM model family. All models share a common base (ModernBERT [35] or mmBERT-32K) and are distributed as LoRA adapters.

Model	**Task**	**Training Data**
`mom-domain`	Domain classification	MMLU categories
`mom-pii`	PII token classification	Presidio-annotated corpora
`mom-jailbreak`	Prompt injection detection	Adversarial prompt datasets
`mom-sentinel`	Fact-check gating	Factual vs. creative queries
`mom-detector`	Hallucination detection	Annotated LLM outputs
`mom-explainer`	NLI explanation	NLI benchmarks
`mom-feedback`	User feedback analysis	Conversation annotations
`mom-modality`	Modality classification	DiffusionDB + text corpora
`mom-embedding`	Semantic embeddings	Contrastive pre-training
`mom-toolcall`	Tool selection	Function-calling datasets
`mom-intent`	User intent classification	Customer support dialogues

The key benefit of distributing these as LoRA adapters rather than independent models is **operational simplicity**: a single base model binary serves all ten tasks, adapters can be hot-swapped without reloading the base, and new tasks can be added by training a new adapter without retraining or redistributing the base model.

8.5 Training Methodology

All LoRA adapters are trained using PEFT [23] with the following protocol:

- **Base model**: ModernBERT or mmBERT-32K (for long-context tasks).
- **Adapter configuration**: Rank $r \in \{16, 32, 64\}$, applied to query and value projection matrices.
- **Training**: Task-specific datasets with standard cross-entropy loss.
- **Export**: Both LoRA-only (separate adapter files for hot-swapping) and merged (single model file for simplified deployment) formats.

The modality classifier, for instance, is trained on a balanced mixture of DiffusionDB (image generation prompts), OASST2, Alpaca, and Dolly (text generation), achieving three-class classification (autoregressive, diffusion, both) with ∼0.02% trainable parameters relative to the base model.

Contributors. Huamin Chen, Yossi Ovadia, Xunzhuo, yehudit1987, abdallahsamabd, Yue Zhu, OneZero-Y, shown, Xunzhuo, asaadbalum, Jared, aeft, Wilson Wu, Tien Nguyen, Srinivas A, Qiping Pan, Nengxing Shen, Liav Weiss, GuanMu, ERIK, Alex Wang, Sanjeev Rampal, Zohaib Hassnain.

9 Semantic Model Selection

The core routing innovation is *semantic model selection*: once the decision engine matches a routing decision d^*, the system analyzes the request's semantic content—its embedding, domain, complexity, and interaction history—to select the most cost-effective model from the decision's candidate set. Unlike static routing or single-criterion difficulty classifiers, semantic selection operates over the full signal context produced by the signal engine, enabling cost-quality optimization that respects per-decision privacy and safety constraints.

We integrate thirteen algorithms within a unified interface, enabling systematic comparison and hybrid combinations across deployment scenarios.

9.1 Problem Setting

Given query embedding $\mathbf{e}_q \in \mathbb{R}^d$, domain category $z \in \{1, \ldots, C\}$, candidate models $\mathcal{M}_{d^*} = \{m_1, \ldots, m_K\}$ with associated costs $\{c_1, \ldots, c_K\}$, and quality estimators, the semantic selection problem is:

$$m^* = \arg \max_{m_k \in \mathcal{M}_{d^*}} \text{Quality}(\mathbf{e}_q, z, m_k; \Theta) - \lambda \cdot \text{Cost}(m_k) \tag{24}$$

where $\lambda \geq 0$ is a cost-sensitivity parameter and Θ represents algorithm-specific parameters. The per-decision candidate set $\mathcal{M}_{d^*}$ is critical: privacy-constrained decisions restrict candidates to compliant models, while cost-optimized decisions include a broader pool with aggressive cost weighting. We categorize algorithms into five families.

9.2 Rating-Based Selection

Static. Each model carries a pre-configured quality score s_k; selection is $m^* = \arg\max_k s_k$. Serves as a deterministic baseline.

Elo Rating (adapted from RouteLLM [26]). Models maintain Elo ratings R_k updated from pairwise user preference feedback. Selection probability follows the Bradley-Terry model:

$$P(m_i \succ m_j) = \frac{1}{1 + 10^{(R_j - R_i)/400}} \tag{25}$$

Models are sampled proportional to their expected win rate against the candidate pool. Ratings are updated online as user feedback arrives.

9.3 Embedding-Based Selection

RouterDC [5]. Dual contrastive learning trains query and model encoders to produce embeddings in a shared space. Selection maximizes cosine similarity:

$$m^* = \arg \max_{m_k \in \mathcal{M}_{d^*}} \cos(\mathbf{e}_q, \mathbf{e}_{m_k}) \tag{26}$$

The contrastive training objective encourages queries to be close to their best-performing model's embedding and distant from poorly-performing models.

Hybrid [10]. Combines Elo ratings, embedding similarity, and cost in a weighted score:

$$\text{score}(m_k) = \alpha \cdot \tilde{R}_k + \beta \cdot \cos(\mathbf{e}_q, \mathbf{e}_{m_k}) + \gamma \cdot (1 - \tilde{c}_k) \tag{27}$$

where $\tilde{R}_k$ and $\tilde{c}_k$ are normalized ratings and costs, and $\alpha + \beta + \gamma = 1$ are configurable weights.

9.4 Cascading Selection

AutoMix [2]. Formulated as a Partially Observable Markov Decision Process (POMDP). Models are ordered by capability $m_1 \prec m_2 \prec \cdots \prec m_K$. The cascade:

1. Generate response a_k with current model m_k (starting from $k = 1$, the cheapest).
2. Self-verify: estimate response quality $\hat{q}_k$ using m_k itself.
3. If $\hat{q}_k \geq \tau_k$, accept a_k; otherwise, escalate to m_{k+1}.

The expected cost is:

$$\mathbb{E}[C] = \sum_{k=1}^{K} C_k \cdot \prod_{j=1}^{k-1} (1 - P(\hat{q}_j \geq \tau_j)) \tag{28}$$

where $P(\hat{q}_j \geq \tau_j)$ is the probability that model m_j passes self-verification. This naturally trades off cost against quality.

9.5 Classical ML Selection

These methods train on routing records $\{(\mathbf{e}_q^{(i)}, z^{(i)}, m^{*(i)}, q^{(i)})\}$ where $q^{(i)}$ is a quality score. Feature vectors combine embeddings and domain information:

$$\mathbf{f} = [\mathbf{e}_q \in \mathbb{R}^d;\ \text{onehot}(z) \in \{0, 1\}^C] \tag{29}$$

KNN. k-nearest neighbor search with Ball Tree indexing. Quality-weighted majority voting:

$$m^* = \arg \max_m \sum_{i \in \text{kNN}(\mathbf{f})} \mathbf{1}[m^{*(i)} = m] \cdot q^{(i)} \tag{30}$$

KMeans. Assigns queries to pre-computed clusters; selects the best model for the assigned cluster based on a combined quality-latency score:

$$m^* = \arg \max_m (\alpha \cdot \text{quality}(m, z_{\text{cluster}}) - (1 - \alpha) \cdot \text{latency}(m)) \tag{31}$$

SVM. Multi-class SVM with RBF or linear kernel, trained to classify feature vectors directly into model selections.

MLP. A feed-forward neural network (two hidden layers with ReLU activation) mapping $\mathbf{f}$ to a softmax distribution over candidate models:

$$P(m_k \mid \mathbf{f}) = \text{softmax}(W_2 \cdot \text{ReLU}(W_1 \mathbf{f} + b_1) + b_2)_k \tag{32}$$

The MLP is implemented in the GPU-accelerated Candle runtime for low-latency inference.

9.6 Reinforcement Learning Selection

RL-Driven (Thompson Sampling). Adapted from Router-R1 [39]. Each model maintains a Beta prior:

$$\theta_k \sim \text{Beta}(\alpha_k, \beta_k) \tag{33}$$

Selection samples from each posterior and picks the maximum: $m^* = \arg\max_k \theta_k$. Parameters (α_k, β_k) are updated from user preference feedback, naturally balancing exploration and exploitation.

GMTRouter [37]. Models multi-turn user-query-model interactions as a heterogeneous graph. Graph neural network message passing captures complex interaction patterns:

$$\mathbf{h}_v^{(l+1)} = \text{AGG}(\{\mathbf{h}_u^{(l)} \mid u \in \mathcal{N}(v)\}) \tag{34}$$

where nodes represent users, queries, and models, and edges encode historical routing outcomes.

9.7 Latency-Aware Selection

Latency-Aware. Selects the model with the best observed latency using percentile-based Time-per-Output-Token (TPOT) and Time-to-First-Token (TTFT) statistics collected at runtime. For each candidate model m_k, the selector computes a normalized latency score:

$$s_k = \frac{1}{|P|} \sum_{p \in P} \frac{\text{perc}_p(m_k)}{\min_j \ \text{perc}_p(m_j)} \tag{35}$$

where $P \subseteq \{\text{TPOT}, \text{TTFT}\}$ is the set of configured performance metrics and $\text{perc}_p(m_k)$ is the observed percentile value for model m_k on metric p. Selection minimizes this score: $m^* = \arg\min_k s_k$. This enables adaptive routing that responds to real-time backend performance degradation without requiring explicit latency thresholds as signal conditions.

9.8 Multi-Round Reasoning (ReMoM)

The ReMoM (Reasoning for Mixture of Models) strategy extends single-shot selection to iterative refinement:

1. **Parallel generation**: Distribute the query to k selected models simultaneously.
2. **Quality assessment**: Score each response using an evaluator.
3. **Synthesis**: Combine responses using quality-weighted aggregation.
4. **Iteration**: Optionally refine with additional rounds using updated model selection.

ReMoM is particularly effective when model capabilities are uncertain or when the task benefits from diverse perspectives (e.g., complex reasoning, multi-faceted analysis).

9.9 Unified Selection Interface

All thirteen algorithms implement a common interface:

$$\text{Select} : (\mathbf{e}_q, z, \mathcal{M}, \Theta) \to (m^*, c) \tag{36}$$

returning the selected model and a confidence score. This uniformity enables: (1) per-decision algorithm selection—different routing decisions can use different selection algorithms, allowing cost-optimized decisions to use cascading (AutoMix) while quality-sensitive decisions use embedding-based (RouterDC) selection; (2) A/B testing across algorithms on live traffic; (3) ensemble methods that combine multiple selectors.

9.10 Cost-Aware Selection in Multi-Provider Settings

In multi-endpoint deployments where the same logical model may be served by different providers at different price points, the selection algorithms operate in conjunction with the endpoint router (Section 11.3). The selection algorithm chooses the best *model* based on semantic analysis, and the endpoint router resolves it to the most cost-effective *provider endpoint*. This two-stage process separates quality optimization (which model is best for this query?) from cost optimization (which provider endpoint offers the best price for this model?), enabling fine-grained cost management across heterogeneous multi-cloud deployments.

10 Multi-Runtime ML Inference

The routing system requires low-latency ML inference for signal extraction, classification, and embedding computation—all on the critical path of every request. We describe the multi-runtime architecture that addresses the tension between inference speed, hardware flexibility, and model diversity.

10.1 Design Constraints

Three constraints shape the inference architecture:

1. **Latency**: Signal extraction must complete within the tail latency budget of the routing system (target: $<100\,\text{ms}$ for all signals combined).
2. **Hardware heterogeneity**: Deployments range from GPU-equipped data centers to CPU-only edge nodes.
3. **Model diversity**: Different tasks require different model architectures (sequence classification, token classification, NLI, embeddings, MLP).

10.2 Three-Runtime Architecture

We implement three inference runtimes, each optimized for different hardware and task profiles, all exposed to the routing layer via C FFI:

Table 3: Inference runtime characteristics

Runtime	Target Hardware	Tasks	Framework
Candle	GPU (CUDA), CPU	Classification, LoRA, MLP	HF Candle [11]
Linfa	CPU only	KNN, KMeans, SVM	Linfa [20]
ONNX RT	CPU, GPU	Embeddings	ONNX Runtime [24]

All runtimes are compiled as Rust shared libraries and linked to the Go routing process via CGo. This eliminates Python runtime overhead, GIL contention, and inter-process communication latency that would arise from serving models in separate Python processes.

10.3 Candle Runtime: GPU-Accelerated Classification

The Candle runtime handles all transformer-based classification tasks, including LoRA adapter loading and inference (Section 8).

Supported architectures. BERT [6], ModernBERT [35] (with Flash Attention and GeGLU), mmBERT-32K (YaRN RoPE for 32K context), DeBERTa v3 (NLI), and feed-forward MLPs (model selection).

Optimization features. Flash Attention 2 kernels reduce attention memory from $O(n^2)$ to $O(n)$ and improve throughput. Optional Intel MKL integration for CPU deployments. LoRA adapter hot-loading enables runtime model updates without restart.

10.4 Linfa Runtime: CPU ML Inference

Classical ML model selection algorithms (KNN, KMeans, SVM) are served by the Linfa runtime. These algorithms operate on pre-computed feature vectors and do not require GPU acceleration, making Linfa's lightweight CPU implementation ideal.

Training-inference split. Models are trained in Python (scikit-learn, custom implementations) and serialized to JSON. The Rust runtime loads serialized models at startup and performs inference-only computation. This decouples the training environment (Python, GPU-optional) from the inference environment (Rust, CPU-only), enabling simpler deployment.

10.5 ONNX Runtime: Efficient Embeddings

Embedding computation is served by ONNX Runtime, optimized for the mmBERT-Embed-32K model with 2D Matryoshka representation learning [16].

2D Matryoshka trade-offs. The architecture supports two-dimensional quality-latency trade-offs:

- **Layer early-exit**: Extract embeddings from intermediate layers (6, 11, 16, or 22 out of 22), achieving $\sim$3–4$\times$ speedup at layer 6 with modest quality degradation.
- **Dimension truncation**: Reduce embedding dimension from 768 to 64, 128, 256, or 512, reducing memory and computation for similarity search.

For the $\sim$150M parameter embedding model, CPU inference with 2D Matryoshka (layer 11, dimension 256) achieves latency comparable to GPU inference on the full model, making GPU optional for embedding computation.

10.6 Runtime Selection Strategy

The routing system selects runtimes based on deployment configuration:

- **GPU available**: Candle (classification + LoRA) + ONNX (embeddings) + Linfa (ML selection).
- **CPU only**: Candle with MKL (classification) + ONNX with early-exit (embeddings) + Linfa (ML selection).
- **Minimal**: ONNX (embeddings) + Linfa (ML selection), with classification delegated to external vLLM-served models.

11 Request Processing Pipeline

We implement the routing system as an Envoy External Processor (ExtProc) [7], enabling transparent interception of LLM API traffic without client-side modifications. This section describes the pipeline architecture, multi-provider routing, the Responses API integration, and the pluggable authorization factory.

11.1 Transparent Interception via ExtProc

The Envoy ExtProc protocol [7] establishes a bidirectional gRPC stream between the proxy and the routing service for each HTTP request. Envoy invokes the processor at four phases—request headers, request body, response headers, response body—and the processor responds with mutations (header modifications, body rewrites) or immediate responses (short-circuiting the backend).

This architecture provides two key advantages: (1) *transparency*: clients send standard OpenAI-compatible API requests to the proxy endpoint with no awareness of the routing layer; and (2) *composability*: the router coexists with other Envoy filters (rate limiting, authentication, load balancing) in the standard filter chain.

11.2 Request Body Pipeline

The request body phase implements the core routing logic as a sequential pipeline:

$$r \xrightarrow{\text{parse}} r' \xrightarrow{\text{signals}} S(r') \xrightarrow{\text{decide}} d^* \xrightarrow{\Pi_{\text{pre}}} \xrightarrow{\text{select}} m^* \xrightarrow{\text{route}} e^* \tag{37}$$

The stages execute in strict order: (1) API translation (Responses API → Chat Completions if applicable, see Section 11.4); (2) request parsing and provider detection; (3) signal extraction and decision evaluation (Sections 3 and 4); (4) jailbreak detection (Section 6); (5) PII detection (Section 6); (6) semantic cache lookup (Section 5)—cache hits terminate the pipeline with an immediate response; (7) RAG context injection (Section 12); (8) modality routing (text vs. diffusion); (9) memory retrieval (Section 12); (10) model selection (Section 9), system prompt injection, and header mutation; (11) multi-endpoint resolution and provider-specific auth injection (Sections 11.3 and 11.5).

11.3 Multi-Endpoint and Multi-Provider Routing

Production deployments often span multiple model backends across different providers and geographic regions. The system supports *multi-endpoint routing* as a first-class concept:

Definition 8 (Endpoint Topology)**.** *An endpoint topology $\mathcal{E} = \{(e_i, w_i, p_i, \alpha_i)\}_{i=1}^{L}$ defines L endpoints, each with a weight $w_i \in (0, 1]$ (normalized: $\sum_i w_i = 1$), a provider type $p_i \in \mathcal{P}$, and an auth profile α_i.*

Once semantic model selection identifies a target model m^*, the endpoint router resolves m^* to a concrete endpoint e^* from the set of endpoints serving that model. Weighted random selection with sticky session affinity distributes load proportionally. Failover cascades to the next-weighted endpoint on backend errors.

Each endpoint may use a different provider (e.g., the same logical model "gpt-4o" served by both OpenAI and Azure OpenAI). The system performs *provider-specific protocol translation* transparently:

- **OpenAI / Azure OpenAI**: Native Chat Completions and Responses API formats.
- **Anthropic**: Translation between OpenAI message schema and Anthropic Messages API (system prompt handling, tool use mapping).
- **Bedrock / Vertex AI**: Cloud-provider-specific request wrapping, authentication (SigV4 for AWS, OAuth for GCP), and response unwrapping.
- **Gemini**: Conversion between OpenAI function-calling schema and Gemini tool declarations.
- **vLLM / Local**: Direct OpenAI-compatible passthrough to self-hosted vLLM instances.

This abstraction allows routing decisions to reference models by capability ("best coding model") rather than by provider-specific endpoint, and allows the same decision configuration to operate across different deployment topologies.

11.4 OpenAI Responses API Support

The system provides full support for the OpenAI Responses API, which extends Chat Completions with stateful multi-turn conversation management.

The Responses API introduces `previous_response_id` chaining: each response carries a unique identifier, and subsequent requests can reference it to maintain conversation context without the client retransmitting the full message history. The routing system handles this by:

1. **Inbound translation**: Responses API requests (with `input` field and `previous_response_id`) are normalized to Chat Completions format for signal extraction and decision evaluation, which operate on the unified internal representation.

2. **State management**: Conversation history is stored in the persistent memory layer (Section 12), keyed by response ID, enabling context retrieval across turns.

3. **Outbound translation**: Chat Completions responses from backends are wrapped in Responses API format (with `id`, `output` array, `usage` breakdown) before returning to the client.

4. **Routing consistency**: The decision engine can optionally pin conversation turns to the same model to avoid mid-conversation quality shifts.

This translation layer is transparent to both the signal engine and the downstream model backends, enabling all routing, safety, and caching features to operate identically on Responses API and Chat Completions traffic.

11.5 Authorization Factory

Multi-provider deployments require diverse authentication mechanisms. The system implements a *pluggable authorization factory* that abstracts auth concerns from routing logic:

Definition 9 (Auth Provider)**.** *An auth provider* $\alpha : (\mathit{Request}, \mathit{Endpoint}) \to \mathit{Headers}'$ *is a function that enriches outbound request headers with provider-appropriate credentials.*

The factory supports multiple auth provider types:

- **API Key**: Static bearer tokens or API keys, optionally per-endpoint, with header name customization (e.g., `Authorization`, `x-api-key`, `api-key`).

- **OAuth2 / OIDC**: Token acquisition with automatic refresh, supporting client credentials and authorization code flows.

- **Cloud IAM**: AWS SigV4 signing for Bedrock, Google service account tokens for Vertex AI, Azure AD tokens for Azure OpenAI.

- **Passthrough**: Forwarding the client's original credentials to the backend, used for deployments where the client authenticates directly.

- **Custom**: User-defined auth plugins registered at startup, enabling integration with enterprise identity providers (LDAP, SAML, custom JWT issuers).

The auth factory is invoked *after* decision evaluation and model selection, injecting provider-specific credentials into the outbound request headers. This separation ensures that routing decisions are auth-agnostic: the decision engine selects models based on capability and cost, and the auth layer handles the mechanics of reaching each provider's endpoint.

The `authz` signal type in the signal engine (Section 3) is complementary but distinct: it performs *inbound* authorization (verifying that the requesting user or API key has permission to access specific models or decisions), while the auth factory handles *outbound* authentication (proving the router's identity to backend providers).

11.6 Response Body Pipeline

The response path performs: (1) token usage extraction for cost accounting; (2) format translation (provider-specific → OpenAI format); (3) streaming metrics computation (TTFT, TPOT); (4) hallucination detection via HaluGate (Section 7); (5) semantic cache writes for cache misses; (6) Responses API translation (Chat Completions → Responses API format, if applicable).

11.7 Concurrency Model

Each gRPC stream (one per HTTP request) runs in an independent goroutine, processing its four phases sequentially. Within a request, signal extraction launches parallel coroutines for independent classifiers. Shared state (classifier models, cache backends, configuration, auth token caches) is read concurrently by all active streams, with synchronization limited to cache writes, auth token refreshes, and metric updates.

12 Memory and Retrieval-Augmented Generation

Production routing systems must support multi-turn conversations with persistent context and knowledge-augmented responses. We describe the memory and RAG subsystems that operate as plugins within the routing pipeline.

12.1 Persistent Memory

The memory system maintains user-scoped knowledge across conversation sessions, enabling personalized routing and context-aware responses. Figure 3 illustrates the full memory lifecycle.

Memory extraction. An LLM-based extractor analyzes conversation turns to identify user-specific facts, classified into three types: *semantic* (factual knowledge: preferences, background), *procedural* (workflows, how-to knowledge), and *episodic* (specific events and interactions).

Deduplication. Before storage, extracted facts undergo similarity-based deduplication against existing memories, preventing redundant entries that would degrade retrieval precision.

Retrieval gating. Not every query benefits from memory retrieval. A lightweight heuristic determines whether memory search is warranted by filtering out general fact-check queries, tool-augmented requests, and simple greetings, avoiding unnecessary embedding lookups and reducing latency for queries where personal context is irrelevant.

Retrieval. At query time, relevant memories are retrieved via embedding similarity search over the user's memory store, formatted as context, and injected into the system message. An optional query-rewriting step reformulates the user's query for improved retrieval recall.

Retention scoring and pruning. Memory stores grow unbounded without lifecycle management. We adopt an exponential decay model inspired by the Ebbinghaus forgetting curve [40]:

$$R = e^{-t/S}, \quad S = S_0 + n_{\text{access}} \tag{38}$$

where t is the time in days since last retrieval, S_0 is the initial strength (default 30 days), and n_{access} is the cumulative access count. Each retrieval reinforces the memory by incrementing n_{access}, slowing future decay. Memories falling below a retention threshold $R < \delta$ become pruning candidates. An optional per-user capacity limit evicts the lowest-scoring entries when the store exceeds a configurable maximum.

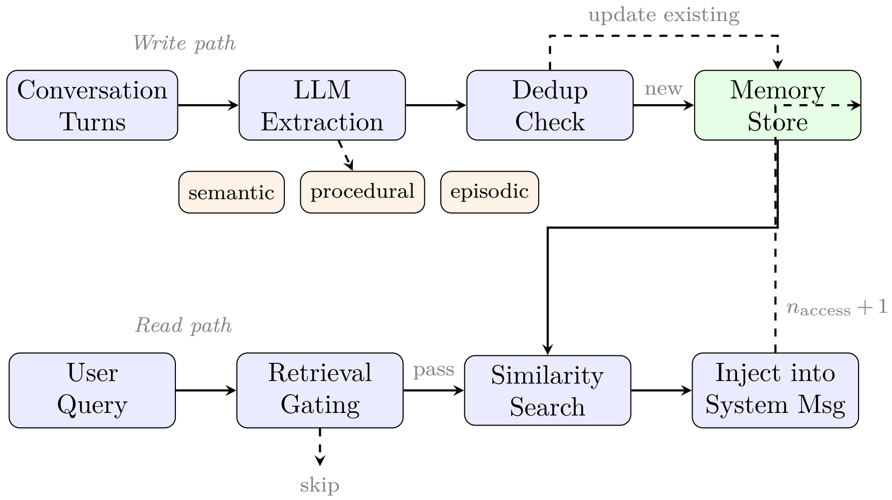

Figure 3: Memory lifecycle. The write path extracts facts from conversations, deduplicates against existing entries, and stores new memories. The read path gates retrieval, searches by embedding similarity, and injects context. Each retrieval increments n_{access}, reinforcing the memory's retention score.

12.2 Retrieval-Augmented Generation

The RAG plugin retrieves relevant documents from vector stores and injects them as context before model invocation.

Indexing pipeline. Documents are chunked (configurable size and overlap), embedded using the shared embedding model (Section 10), and indexed in a vector store.

Retrieval. For each request, the query embedding is computed and the top-k most similar chunks are retrieved via approximate nearest neighbor search. Retrieved chunks are formatted and injected into the prompt as additional context.

Backend abstraction. Vector stores are accessed through a common interface supporting in-memory (development), Milvus [34] (production), and file-based (persistent, no external dependencies) backends.

12.3 Stateful Conversations (Response API)

The system supports the OpenAI Responses API for stateful multi-turn conversations:

Conversation chaining. Each response is stored with a unique ID. Subsequent requests reference `previous_response_id` to reconstruct the full conversation history. A bidirectional translator converts between the Response API format and Chat Completions format for backend model invocation.

Routing continuity. Stored responses include routing metadata (decision, model selection, signal results), enabling consistent routing across conversation turns and providing context for feedback-driven model selection.

State backends. Conversation state is persisted via in-memory (development), Redis (distributed), or Milvus (semantic retrieval) backends.

12.4 Integration with Signal-Decision Architecture

Both memory and RAG operate as per-decision plugins. Different decisions can activate different RAG configurations (different vector stores, different k values, different chunk strategies) or disable retrieval entirely. This enables, for example, a "research assistant" decision that activates RAG with a technical knowledge base while a "casual chat" decision disables retrieval.

13 Observability

Intelligent routing introduces a new observability surface: beyond standard model serving metrics, operators must monitor signal extraction quality, decision matching patterns, plugin effectiveness, and model selection outcomes.

13.1 Metrics Taxonomy

We instrument four metric categories using Prometheus [28]:

Model performance. Per-model request counts, token usage, estimated cost, completion latency, Time-to-First-Token (TTFT), and Time-per-Output-Token (TPOT). These metrics enable real-time cost monitoring and performance regression detection across the model fleet.

Routing behavior. Routing modification counts (original model vs. selected model), reason codes (which signal types triggered which decisions), and routing latency (overhead of the routing pipeline itself). These metrics answer the question: *how is the router changing traffic patterns?*

Signal and decision quality. Per-signal-type extraction counts and match rates, per-decision match frequencies and confidence distributions, and per-plugin execution counts and outcomes. These metrics enable calibration: if a signal type rarely matches, its threshold may need adjustment; if a decision matches too broadly, its conditions may be under-specified.

Safety and cache effectiveness. PII violation rates by entity type, jailbreak detection rates, hallucination detection latency, cache hit rates, and cache operation latency. These metrics quantify the value delivered by the plugin chain.

13.2 Distributed Tracing

We implement OpenTelemetry [27] tracing with a hierarchical span model:

- **Root span**: Covers the full request lifecycle from receipt to response.
- **Signal spans**: Individual spans for each signal type evaluation, capturing latency and results.
- **Decision span**: Decision evaluation with the matched decision and confidence.
- **Plugin spans**: Per-plugin execution with type-specific attributes (cache hit/miss, PII types detected, hallucination spans found).
- **Upstream span**: Backend model invocation, with W3C Trace Context propagation enabling end-to-end tracing through vLLM [17] and other inference frameworks.

This span hierarchy enables operators to diagnose routing latency ("which signal is slow?"), understand routing decisions ("why was this query routed to model X?"), and correlate routing behavior with model serving performance.

14 Deployment

We describe the deployment architecture that enables the routing system to operate from single-node development to production Kubernetes clusters.

14.1 Deployment Modes

Local development. A single command (`pip install vllm-sr && vllm-sr serve`) bootstraps the complete stack: router, Envoy proxy, and dashboard. This lowers the barrier to experimentation with routing configurations.

Kubernetes: Standalone mode. Envoy runs as a sidecar container alongside the router in the same pod. The ExtProc filter connects via localhost gRPC. Deployed via Helm charts with configurable replicas, resource limits, cache backends, and autoscaling.

Kubernetes: Gateway mode. The router runs as an independent service behind an existing Istio or Envoy Gateway deployment, referenced via the gateway's ExtProc configuration. This mode shares the gateway infrastructure across multiple services.

14.2 Kubernetes Operator

A custom operator manages the lifecycle of routing deployments through a `SemanticRouter` Custom Resource Definition (CRD). The reconciliation loop manages: service accounts, configuration (ConfigMap or CRD-sourced), persistent storage, gateway/route resources, Envoy configuration, deployments, services, and horizontal pod autoscalers.

Backend discovery. The operator discovers model backends via three mechanisms: KServe InferenceService resources (for managed model serving), label-based Llama Stack discovery, and direct Kubernetes Service references.

14.3 Dashboard

A web console (React frontend, Go backend) provides: configuration editing with live validation, topology visualization of routing flows, an interactive playground for testing routing decisions, embedded Grafana/Prometheus/Jaeger dashboards for monitoring and tracing, and an evaluation framework for benchmarking routing quality.

15 Evaluation

We evaluate the routing system across three dimensions: signal extraction efficiency, LoRA multi-task scaling, and end-to-end routing correctness.

15.1 Signal Extraction Latency

Table 4 reports median and p99 latencies for each signal type on an NVIDIA A100 GPU with ModernBERT base model.

Heuristic signals complete in sub-millisecond time, while ML signals range from 15–120 ms. With parallel evaluation, the wall-clock time is dominated by the slowest active signal ($\sim$120 ms for domain classification) rather than the sum.

15.2 LoRA Memory Efficiency

Table 5 shows the memory advantage of serving classifiers via LoRA adapters versus independent fine-tuned models.

Table 4: Signal extraction latency by type

Signal Type	Median	p99	Requires ML
Keyword	< 0.1 ms	< 0.5 ms	No
Context	< 0.1 ms	< 0.5 ms	No
Language	< 0.5 ms	< 1 ms	No
Authorization	< 0.1 ms	< 0.5 ms	No
Embedding	15 ms	45 ms	Yes
Domain	60 ms	120 ms	Yes
Fact-check	55 ms	110 ms	Yes
Modality	50 ms	100 ms	Yes
Feedback	55 ms	115 ms	Yes
Complexity	50 ms	105 ms	Yes
Preference	55 ms	110 ms	Yes

Table 5: Model memory: independent fine-tuned models vs. LoRA adapters (ModernBERT base, 150M params)

Tasks (n)	Independent (MB)	LoRA (MB)
1	573	573
3	1,719	574
6	3,438	575

At $n = 6$, the LoRA architecture requires $\sim$6× less model memory (one base model + six $\sim$0.2 MB adapters vs. six full model copies). Each task still requires its own forward pass; latency reduction comes from *parallel execution* of classifiers rather than from LoRA itself (Section 8).

15.3 Decision Engine Overhead

Decision evaluation adds negligible latency: < 0.1 ms for 10 decisions with 3 conditions each; < 0.5 ms for 100 decisions with 5 conditions each. This confirms that the $O(M \cdot L_{\max})$ complexity is dominated by signal extraction.

15.4 Composable Orchestration Across Deployment Scenarios

A key claim of this work is that the same architecture serves diverse deployment scenarios through configuration. Table 6 demonstrates how different signal-decision-plugin compositions address different requirements:

15.5 End-to-End Routing Correctness

The end-to-end test framework validates routing behavior across eight scenario profiles:

Each profile validates correct model selection, safety enforcement (jailbreak blocked, PII detected), cache behavior (hits after similar queries), multi-provider routing (correct endpoint resolution and auth injection), and header propagation.

15.6 Semantic Cache Effectiveness

At a similarity threshold $\theta = 0.92$: exact-match queries achieve 100% hit rate with < 5 ms lookup latency; paraphrased queries achieve 60–80% hit rate depending on paraphrase distance.

Table 6: Composable signal orchestration across deployment scenarios. Each scenario activates a different subset of the eleven signal types, selection algorithms, and plugin chains—using the same system binary and architecture.

Scenario	**Active Signals**	**Selection**	**Key Plugins**
Privacy-regulated (healthcare)	authz, domain, language	Static (compliant models only)	Strict PII redaction, no caching, audit logging
Cost-optimized (developer tool)	complexity, embedding, keyword	AutoMix cascade	Aggressive semantic cache, header mutation for LoRA adapter
Multi-cloud enterprise	domain, modality, authz	Latency-aware	Multi-endpoint failover, provider auth factory, system prompt injection
Multi-turn assistant	embedding, feedback, preference	Elo with session pin	Responses API state, memory retrieval, RAG injection

Cache hits eliminate backend model invocation entirely, reducing per-request cost to embedding computation only.

15.7 Unified MoM Evaluation Framework

To validate the robustness of the Mixture of Models (MoM) collection, we implemented a unified evaluation pipeline that benchmarks both merged models and LoRA adapters. The framework standardizes the assessment of heterogeneous model variants across intent classification and PII detection tasks.

The evaluation architecture, shown in Figure 4, utilizes the following components:

- **Schema Normalization:** Inputs from diverse sources—including MMLU-Pro for intent and Presidio for token classification—are mapped to a common evaluation schema.
- **Comparative Quality Validation:** The system computes weighted F1-scores and per-class precision/recall to ensure that the memory efficiency gains reported in Table 5 do not result in significant predictive degradation compared to full-parameter models.
- **Parallelized Benchmarking:** Large-scale evaluations are executed via `ProcessPoolExecutor` to minimize wall-clock time. The pipeline incorporates automated OOM recovery and exponential backoff for API-based signal providers to ensure benchmark reliability.

Results include p50 and p99 latency profiling to confirm that model orchestration remains within the operational bounds required for real-time routing.

15.8 Open Evaluation

Detailed model selection quality comparisons across algorithms, HaluGate detection accuracy on standard benchmarks (HaluEval, FActScore), and large-scale routing quality evaluation are under preparation in collaboration with the RouterArena team.

Table 7: End-to-end test profiles

Profile	Validated Behavior
Multi-endpoint	Multi-provider routing with weighted distribution and failover across heterogeneous backends
Multi-provider auth	Provider-specific auth injection (API key, OAuth2, cloud IAM) via authorization factory
AuthZ-RBAC	Role-based model access (admin/premium/free tiers) with authz signal
ML model selection	KNN, KMeans, SVM, MLP selection accuracy on held-out queries
Keyword routing	Keyword signal matching with AND/OR/NOR combinators
Embedding routing	Embedding similarity thresholds and confidence-based decision selection
RAG + Responses API	Context retrieval, injection, and stateful multi-turn via Responses API
Routing strategies	Static, Elo, RouterDC, AutoMix, Hybrid algorithm comparison

Contributors. Senan Zedan, Xunzhuo, yehudit1987, abdallahsamabd, Noa Limoy, Yossi Ovadia, Liav Weiss, Huamin Chen, samzong, Jintao Zhang, asaadbalum, Xunzhuo, Srinivas A, Marina Koushnir, Chever John, Chaojun Zhang, Avishek Goswami, Zohaib Hassnain.

16 Related Work

16.1 LLM Routing and Model Selection

Binary routing. RouteLLM [26] pioneered preference-data-driven routing between a strong and weak model, training BERT, MLP, and causal LLM classifiers to estimate query difficulty. Our work extends this to multi-model, multi-signal routing with per-decision plugin chains.

Contrastive selection. RouterDC [5] learns shared query-model embeddings via dual contrastive learning. We integrate RouterDC as one of thirteen selection algorithms and extend it with signal-conditioned features (domain category, complexity).

Cascading. AutoMix [2] formulates model cascading as a POMDP with self-verification. We integrate AutoMix within our plugin-aware framework, where safety checks and caching can prevent unnecessary escalation.

Benchmarking. RouterBench [10] proposed a benchmark for multi-LLM routing with hybrid scoring. Our Hybrid selector builds on this approach.

RL-based routing. Router-R1 [39] applies reinforcement learning with Thompson sampling for multi-round routing. GMTRouter [37] uses graph-based learning for personalized multi-turn interactions. We integrate both within the unified selection interface and extend them with the ReMoM multi-round reasoning strategy.

A key distinction of our work is that prior approaches address model selection in isolation, while we embed selection within a composable signal orchestration framework that also handles signal extraction, safety enforcement, caching, context augmentation, and multi-provider routing—enabling the same selection algorithms to serve fundamentally different deployment scenarios through configuration.

16.2 Multi-Provider and Multi-Endpoint Routing

Commercial LLM gateway products (OpenRouter, AWS Bedrock, Azure AI Studio) provide multi-provider access but lack the composable signal-driven routing that enables differentiated policies per routing decision. API management platforms (Kong, Apigee) offer gateway functionality but are not designed for semantic analysis of LLM requests. Our system uniquely combines semantic model selection with multi-provider protocol abstraction, a pluggable authorization factory, and full OpenAI Responses API support for stateful conversations within the same composable framework.

16.3 Mixture-of-Experts vs. Mixture-of-Models

Sparse Mixture-of-Experts (MoE) [33, 15] routes *tokens* to specialized sub-networks *within* a single model architecture. Our system operates at the *request level*, routing entire requests across *different model deployments*—a Mixture-of-Models (MoM) approach. The two paradigms are complementary: our router can route to MoE models as backends.

16.4 LLM Safety

Prompt injection defenses [14, 13] detect adversarial inputs via fine-tuned classifiers. PII detection systems [21] identify sensitive information using rule-based and ML approaches. Our safety subsystem integrates both within the routing pipeline with per-decision thresholds and policies, using LoRA adapters for memory-efficient multi-task classification.

16.5 Hallucination Detection

SelfCheckGPT [22] detects hallucinations via multi-sample consistency. FActScore [25] evaluates factual precision at the atomic fact level. HaluGate differs in three respects: (1) a gating Sentinel that skips verification for non-factual queries; (2) token-level span identification rather than sentence-level scoring; (3) NLI-based explanation distinguishing contradiction from neutral unsupported content.

16.6 Semantic Caching and RAG

Semantic caching for LLMs [18] uses embedding similarity for query matching. Our cache extends this with per-decision policies, multiple backends, and integration with the safety pipeline (cache lookups occur *after* safety checks but *before* model invocation). RAG integration [18, 34] augments responses with retrieved context; our contribution is embedding RAG as a per-decision plugin within the routing framework.

17 Conclusion

We have presented vLLM Semantic Router, a signal-driven decision routing system for Mixture-of-Modality model deployments. The central contribution is **composable signal orchestration**: the three-layer architecture—signal extraction, Boolean decision evaluation, per-decision plugin chains—enables diverse deployment scenarios to be expressed as different configurations over the same framework, without code changes.

Privacy-regulated deployments activate authz and PII signals with strict filtering plugins; cost-optimized deployments enable cascading selection with aggressive semantic caching; multi-cloud enterprises configure weighted multi-endpoint routing with provider-specific auth injection. All use the same signal-decision-plugin machinery, composed differently.

Within this framework, **semantic model selection** analyzes each request's content through thirteen algorithms—spanning rating-based, contrastive, cascading, classical ML, reinforcement learning, and latency-aware families—to find the most cost-effective model while respecting per-decision privacy and safety constraints. The integration of full **OpenAI Responses API support** enables stateful multi-turn routing with conversation-consistent model assignment; **multi-endpoint and multi-provider routing** abstracts over heterogeneous backends (vLLM, OpenAI, Anthropic, Azure, Bedrock, Gemini, Vertex AI) with transparent protocol translation; and the **pluggable authorization factory** supports diverse auth mechanisms across providers without coupling auth logic to routing decisions.

Additional technical contributions include: (1) the LoRA-based multi-task classification architecture that serves n classifiers from a single base model, reducing aggregate model memory by $\sim n\times$; (2) HaluGate's gated three-stage hallucination detection pipeline that reduces average detection cost by $\sim$50% through sentinel-based filtering; and (3) Rust-native ML inference bindings (Candle, Linfa, ONNX Runtime) that achieve sub-10 ms signal extraction latency.

The system has been validated in production with over 600 merged contributions from 50+ engineers and is deployed as an Envoy ExtProc with Kubernetes operator support.

17.1 Future Directions

Several research directions emerge from this work:

Learned decision policies. Replacing hand-crafted Boolean rules with learned routing policies (e.g., neural routing networks trained on production traffic) could improve routing quality while maintaining interpretability through attention-based explanation.

Adaptive cost optimization. Online learning approaches that continuously adapt model and provider selection based on real-time cost signals, latency measurements, and user feedback—extending the current offline-trained ML selectors to fully adaptive cost-quality optimization.

Cross-provider consistency. Techniques for ensuring consistent behavior when routing the same conversation across different providers, addressing differences in instruction following, safety behavior, and output formatting.

Multi-turn safety. Extending safety enforcement from single-turn to multi-turn conversations, detecting adversarial patterns that span multiple interaction rounds.

Federated signal orchestration. Extending composable signal orchestration to federated deployments where signals from multiple routing instances are aggregated for global optimization.

Multi-protocol adapter abstraction. The current system is tightly coupled to Envoy's External Processor protocol. A multi-protocol adapter architecture would abstract the routing engine from protocol-specific code, enabling support for HTTP REST, native gRPC, Nginx/OpenResty, and custom protocols through thin translation layers. This would also enable abstraction of backend proxying (currently Envoy-specific), external authorization mechanisms, and traffic management policies, making the routing engine truly protocol-agnostic and deployable in serverless, edge, and non-Envoy environments.

Acknowledgments

We gratefully acknowledge contributions who contributed to the project. We thank the Hugging Face Candle team for collaboration on the Candle inference runtime, and the Envoy community for the ExtProc filter. AMD has sponsored the project with resources and infrastructure as well as the vLLM project for the development of the project. The project is open-source at https://github.com/vllm-project/semantic-router.

References

[1] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers.* Princeton University Press, Princeton, NJ, 2008.

[2] Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. AutoMix: Automatically mixing language models. *arXiv preprint arXiv:2310.12963*, 2023.

[3] Richard E. Bellman and Lotfi A. Zadeh. Decision-making in a fuzzy environment. *Management Science*, 17(4):B–141–B–164, 1970.

[4] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, MA, 1984.

[5] Shuhao Chen, Weisen Jiang, Baijiong Lin, James T. Kwok, and Yu Zhang. RouterDC: Query-based router by dual contrastive learning for assembling large language models. *arXiv preprint arXiv:2409.19886*, 2024.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2019.

[7] Envoy Proxy Authors. Envoy external processing filter, 2024.

[8] Harvey Fleisher and Lester I. Maissel. An introduction to array logic. *IBM Journal of Research and Development*, 19(2):98–109, 1975.

[9] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2022.

[10] Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. RouterBench: A benchmark for multi-LLM routing system. *arXiv preprint arXiv:2403.12031*, 2024.

[11] Hugging Face. Candle: Minimalist ML framework for Rust, 2024.

[12] Edward V. Huntington. Sets of independent postulates for the algebra of logic. *Transactions of the American Mathematical Society*, 5(3):288–309, 1904.

[13] Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, and Madian Khabsa. Llama Guard: LLM-based input-output safeguard for human-AI conversations. *arXiv preprint arXiv:2312.06674*, 2023.

[14] Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. Baseline defenses for adversarial attacks against aligned language models. *arXiv preprint arXiv:2309.00614*, 2023.

[15] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[16] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. Matryoshka representation learning. *arXiv preprint arXiv:2205.13147*, 2022.

[17] Woosuk Kwon, Zhuohan Li, Sicheng Zuo, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[18] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. *arXiv preprint arXiv:2005.11401*, 2020.

[19] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, pages 661–670. ACM, 2010.

[20] Linfa Contributors. Linfa: A Rust machine learning framework, 2024.

[21] Pierre Lison, Ildikó Pilán, David Sánchez, Montserrat Batet, and Lilja Øvrelid. Anonymisation models for text data: State of the art, challenges and future directions. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 4188–4203, 2021.

[22] Potsawee Manakul, Adian Liusie, and Mark J.F̃. Gales. SelfCheckGPT: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*, 2023.

[23] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. PEFT: State-of-the-art parameter-efficient fine-tuning methods. In *Hugging Face*, 2022.

[24] Microsoft. ONNX Runtime: Cross-platform, high performance ML inferencing and training accelerator, 2024.

[25] Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. *arXiv preprint arXiv:2305.14251*, 2023.

[26] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[27] OpenTelemetry Authors. OpenTelemetry: An observability framework for cloud-native software, 2024.

[28] Prometheus Authors. Prometheus: From metrics to insight, 2024.

[29] Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[30] Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[31] Shai Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2):107–194, 2012.

[32] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[33] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[34] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021.

[35] Benjamin Warner, Antoine Chaffin, Benjamin Clavie, Orion Weller, Oskar Hallstrom, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context fine-tuning and inference. *arXiv preprint arXiv:2412.13663*, 2024.

[36] Adina Williams, Nikita Nangia, and Samuel R. Bowman. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*, 2018.

[37] Encheng Xie, Yihang Sun, Tao Feng, and Jiaxuan You. GMTRouter: Personalized LLM router over multi-turn user interactions. *arXiv preprint arXiv:2511.08590*, 2025.

[38] Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

[39] Haozhen Zhang, Tao Feng, and Jiaxuan You. Router-R1: Teaching LLMs multi-round routing and aggregation via reinforcement learning. *arXiv preprint arXiv:2506.09033*, 2025.

[40] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. *arXiv preprint arXiv:2305.10250*, 2023.

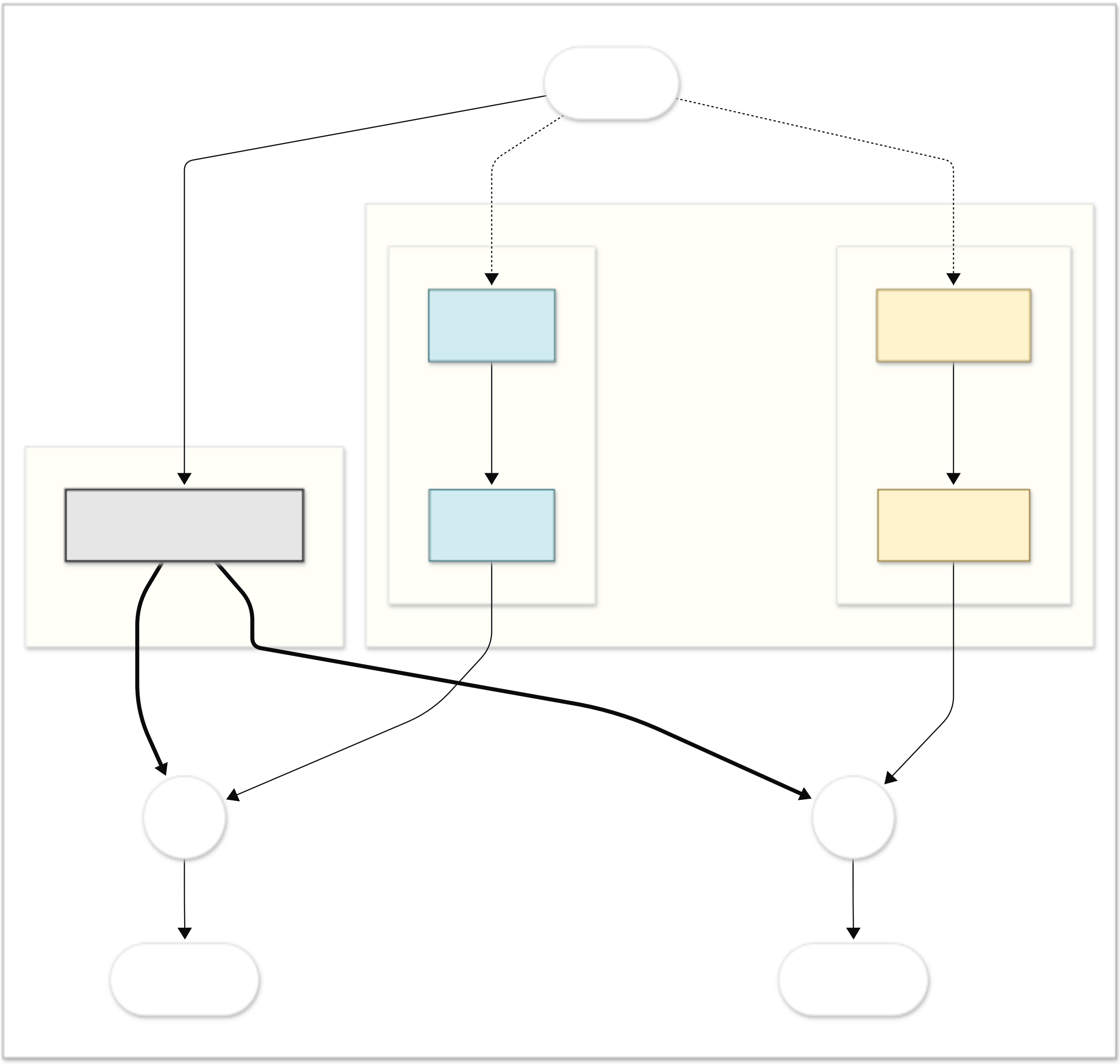

Figure 2: LoRA-based MoM Inference Architecture. The input query is processed by a single, shared frozen base model (W). Simultaneously, task-specific low-rank adapters ($A \times B$) calculate perturbations. The outputs are summed element-wise to produce final task-specific classifications, enabling multi-task support with minimal memory overhead.

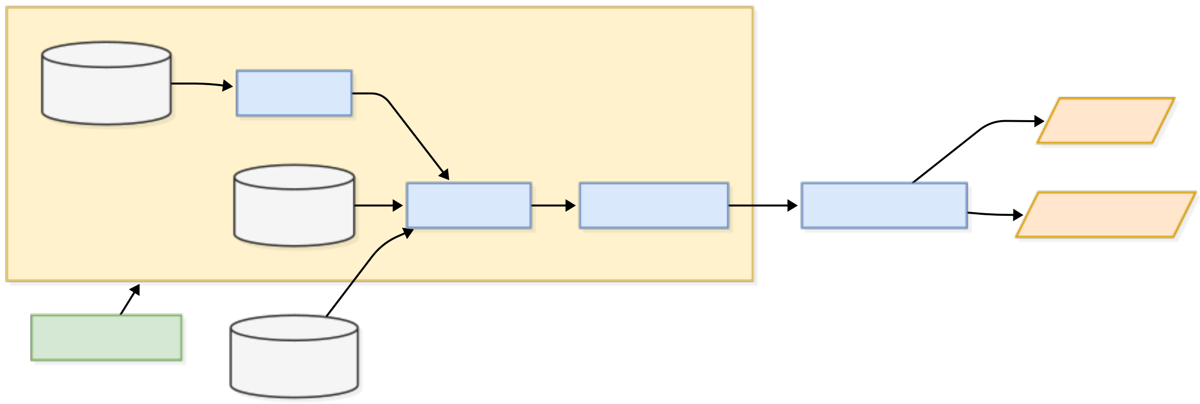

Figure 4: Unified MoM evaluation pipeline. The system parallelizes assessment across the model registry, managing dynamic dataset loading and comparative metric generation for merged and LoRA variants.