vLLM Semantic Router:

Signal Driven Decision Routing for Mixture-of-Modality Models

Xunzhuo Liu, Huamin Chen, Samzong Lu, Yossi Ovadia, Guohong Wen, Hao Wu, Zhengda Tan, Jintao Zhang, Senan Zedan, Yehudit Kerido, Liav Weiss, Haichen Zhang, Bishen Yu, Asaad Balum, Noa Limoy, Abdallah Samara, Baofa Fan, Brent Salisbury, Ryan Cook, Zhijie Wang, Qiping Pan, Rehan Khan, Avishek Goswami, Houston H. Zhang, Shuyi Wang, Ziang Tang, Fang Han, Zohaib Hassan, Jianqiao Zheng, Avinash Changrani

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 architecture rests on a theoretical foundation that mirrors two pillars of Shannon's work: the signal extraction layer operates in the probabilistic regime of information theory—reducing the entropy of "which model?" by extracting routing-relevant information from raw queries—while the decision engine operates in the algebraic regime of Boolean logic synthesis, composing functionally complete routing policies from signal conditions exactly as Shannon's switching algebra composes circuit behavior from relay states.

The central innovation is *composable signal orchestration*: thirteen heterogeneous signal types—spanning sub-millisecond heuristics and neural classifiers for semantics, safety, and modality—are composed through configurable Boolean decision rules into deployment-specific routing policies, so that fundamentally different scenarios (multi-cloud enterprise, privacy-regulated, cost-optimized) are expressed as different configurations over the same architecture.

Matched decisions drive *semantic model routing* via thirteen selection algorithms, while per-decision plugin chains enforce safety constraints including a three-stage *HaluGate* hallucination detection pipeline and a lightweight episodic memory system with *ReflectionGate* for personalized multi-turn context. Deployed in production as an Envoy external processor with multi-provider routing across heterogeneous backends, the architecture demonstrates that composable signal orchestration enables a single 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 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?*

Corresponding repository: GitHub repository

Viewed through the lens of information theory Shannon (1948), routing is an *uncertainty-reduction* problem. Before any analysis, routing entropy is maximal: $H(M \mid r_{\text{raw}}) \approx \log_2 K$ bits for K candidate models—every model is equally plausible. Each signal extracted from the request reduces this uncertainty.

The objective is to collapse $H(M \mid S(r))$ to near zero so the decision engine can make a deterministic, high-confidence routing choice (Figure 2). This viewpoint suggests a natural two-layer decomposition (Figure 1).
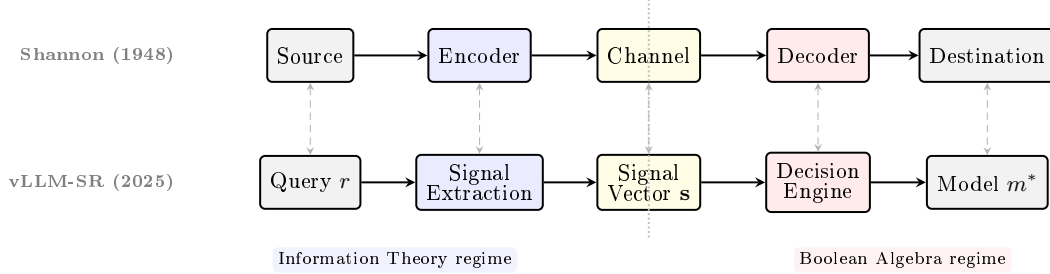

Figure 1: Structural correspondence between Shannon's communication system Shannon (1948) and VSR's routing pipeline. The dotted line marks the boundary between the information-theoretic regime (signal extraction maximizes mutual information with the routing outcome) and the Boolean-algebraic regime (the decision engine synthesizes routing policies from signal conditions via $\{\wedge, \vee, \neg\}$ Shannon (1938)).

In the first layer, signal extraction operates in the probabilistic regime of Shannon's *Mathematical Theory of Communication*; in the second, decision evaluation operates in the algebraic regime of Shannon's *switching-circuit algebra* Shannon (1938). The signal vector **s** is the interface between these regimes: continuous probabilistic inference below, discrete symbolic logic above. In modern ML terms, the extraction layer behaves like a *hybrid embedding stage* (Section 3.8), while the priority-ordered decision engine behaves like a *symbolic Mixture-of-Experts gate* with deterministic early exit (Section 4.8). We further view priority blocks as *layered entropy folding* (Section 4.9), yielding a programmable neural-symbolic inference engine (Transformer-like control structure but not equivalent hidden-state dynamics) that we connect to agent-based policy synthesis (Section 6.8).
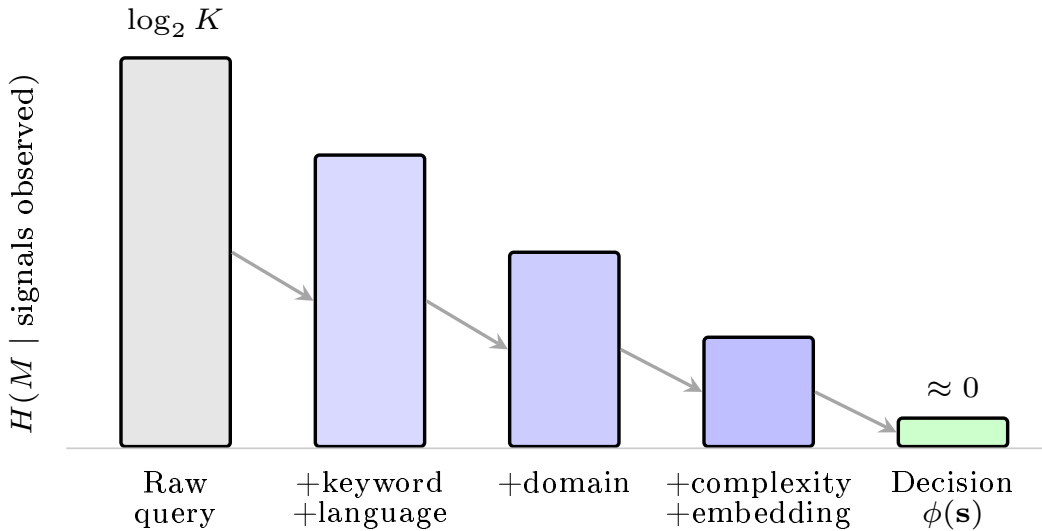

Figure 2: Entropy collapse during signal extraction. Each additional signal reduces the routing uncertainty $H(M \mid \cdot)$ from the uniform prior $\log_2 K$ bits until the decision formula $\phi(\mathbf{s})$ yields a near-deterministic model selection.

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, 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 Ong et al. (2024) trains classifiers to route between two models based on query difficulty. RouterDC Chen et al. (2024) learns query-model embeddings via dual contrastive learning. AutoMix Aggarwal et al. (2023) 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 thirteen 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 10): 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 8): 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 12): 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 9 and 11): 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.

6. **Episodic Conversation Memory with ReflectionGate** (Section 13): A lightweight memory system that stores raw conversational turns as episodic chunks (filtered by an entropy gate and capped at 16 KB) rather than relying on LLM-based fact extraction, eliminating inference overhead at write time. At retrieval time, a multi-stage *Reflection-Gate* pipeline—safety block-pattern filtering, recency decay, Jaccard deduplication, and budget capping—refines retrieved chunks before injection as a separate context message, enabling personalized multi-turn routing without coupling memory quality to an external model.

7. **Programmable Neural-Symbolic Configuration Language** (Section 6): A typed configuration language that serves as the instruction set of the routing inference engine, with a formal grammar parsed into a Boolean expression AST, three-level validation (syntax, reference, constraint), multi-target compilation (flat YAML, Kubernetes CRDs, Helm charts), and round-trip decompilation. We formalize the system as a *programmable neural-symbolic inference engine*—neural signal extraction as a hybrid embedding layer, symbolic decision evaluation as Mixture-of-Experts gating—and show that the language's functional completeness enables LLM-based coding agents to synthesize routing policies from natural-language specifications.

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, 7 and 8 describe the plugin framework and safety subsystems. Sections 9 and 11 detail the LoRA-based classification architecture and multi-runtime inference design. Section 10 surveys the semantic model selection algorithms. Section 12 describes the multi-provider request processing pipeline. Sections 13 to 15 cover memory, observability, and deployment. Section 16 presents evaluation results. Section 6 specifies the programmable configuration language and develops the neural-symbolic inference engine perspective. Section 17 discusses related work, and Section 18 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
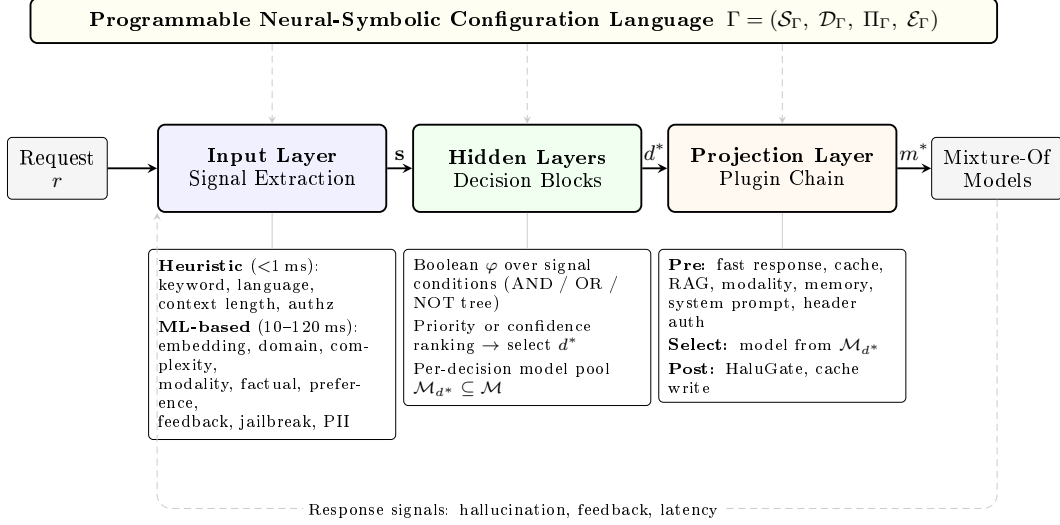
Figure 3: Three-layer architecture with closed-loop feedback. A deployment configuration Γ selects which signals, decisions, and plugins are active. Layer 1 extracts a signal vector $\mathbf{s}$ from the request. Layer 2 evaluates Boolean decision formulas to select d^*. Layer 3 executes the per-decision 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.

strategies, prompt augmentation, model pools) for different routing outcomes—and that these policies must be *composable* to support diverse deployment scenarios without architectural changes.

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 (Programmable Neural-Symbolic Configuration Language)**.** *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 3):

Input Layer: 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 thirteen 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, jailbreak detection, PII detection) 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.

Hidden Layers: Decision Blocks. 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).

Projection Layer: Plugin Chain. Each decision d^* carries a per-decision plugin configuration that defines: (a) *pre-routing plugins* (fast response for safety enforcement, 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 thirteen signal types are the built-in set; the framework is designed to be extended with domain-specific signals as deployment requirements evolve. This strict decoupling has a theoretical antecedent in Shannon's *source-channel separation theorem* Shannon (1948): just as a communication system can be decomposed into a source encoder (which compresses the message to its informational essence) and a channel encoder (which adds structure for reliable transmission)—each optimized independently without loss of optimality—the routing system decomposes into a *signal extraction layer* (which distills routing-relevant information from the raw request) and a *decision layer* (which synthesizes a routing policy from the extracted signals). The signal vector $\mathbf{s}$ is the interface between these independently optimizable stages, analogous to the compressed source representation at the encoder–channel boundary. This separation also mirrors the architecture of Mixture-of-Experts models Shazeer et al. (2017), where the gating network (signal extraction) and expert modules (model backends) are designed independently—though our gating is symbolic rather than neural, enabling formal verification and compositional editing (Section 4.8).

Closed-loop adaptivity. The bidirectional signal flow described in Section 3 enables the architecture to operate as a *closed-loop control system* Åström & Murray (2008). 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,

6

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 Li et al. (2010): 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 Shalev-Shwartz (2012) 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 omit PII signal conditions while a customer-support decision enforces strict PII filtering via signal-matched fast response—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 thirteen orthogonal dimensions. Figure 4 provides an overview of the signal taxonomy and evaluation flow. 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:*

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

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

The thirteen 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.
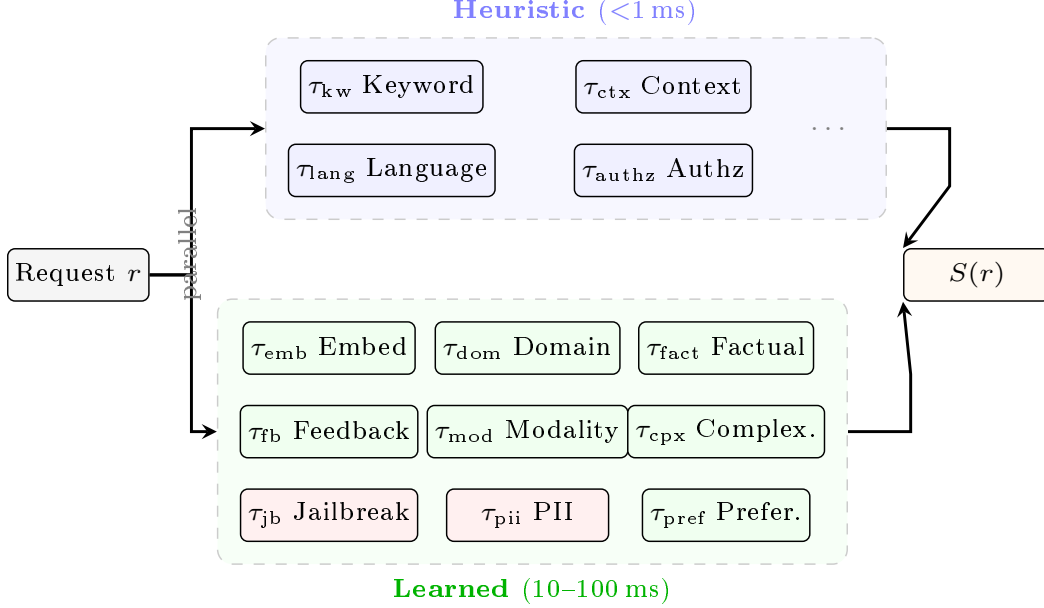
Figure 4: Signal extraction taxonomy and evaluation flow. An incoming request is evaluated in parallel against heuristic signals (sub-millisecond, deterministic) and learned signals (neural inference via LoRA classifiers). Only signal types referenced by configured decisions are computed (demand-driven evaluation). Results merge into the structured signal result $S(r)$.

For AND: $f(r) = \bigwedge_i \mathrm{match}(p_i, r)$; for OR: $f(r) = \bigvee_i \mathrm{match}(p_i, r)$. 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 12.5) 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 9.

Why encoder-based models. The choice of bidirectional encoders (ModernBERT Warner et al. (2024), mmBERT-32K for long-context tasks) rather than unidirectional decoders for signal extraction is not merely an efficiency decision—it reflects a fundamental information-theoretic principle. Routing requires *understanding* the query: determining its domain, complexity, intent, modality, and the precise location of sensitive content. Understanding,

in Shannon's framework Shannon (1948), corresponds to building representations that maximize the mutual information $I(\mathbf{H}; Y)$ between the hidden states $\mathbf{H}$ and the task label Y. A bidirectional encoder conditions every token on the full context in both directions, producing hidden states that capture the complete information structure of the input; a causal decoder, by contrast, conditions each token only on its left context—its representations are optimized for next-token prediction $I(\mathbf{h}_t; x_{t+1})$, which captures *generative* continuation rather than *discriminative* understanding.

This bidirectional "understanding" is exploited at three distinct granularities across the signal types:

- **Sequence-level** (CLS pooling): For domain classification, jailbreak detection, fact-check gating, modality classification, and user feedback, the pooled CLS representation acts as an approximate *sufficient statistic*—a fixed-dimensional vector that compresses the query's global semantics while discarding positional detail irrelevant to the label.
- **Token-level** (per-token hidden states): For PII detection and hallucination span identification, the *position* of information matters—which tokens are names, which spans are unsupported claims. Here the full sequence of hidden states $(\mathbf{h}_1, \ldots, \mathbf{h}_T)$ is retained, preserving the positional mutual information $I(\mathbf{h}_t; y_t)$ that CLS pooling would discard.
- **Cross-sequence** (cross-encoder): For NLI-based hallucination explanation (Section 8), the encoder jointly attends over a (claim, evidence) pair, maximizing the *inter-sequence* mutual information $I(\text{claim}; \text{evidence})$ through full cross-attention—a capacity unavailable to architectures that encode each sequence independently.

In each case, the encoder's unrestricted attention pattern—the absence of a causal mask—is what enables maximal information extraction at the granularity the task demands.

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 8) 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}). A contrastive embedding classifier estimates query difficulty. Each complexity rule defines two sets of candidate exemplars—*hard* (e.g., multi-step reasoning problems) and *easy* (e.g., simple factual lookups)—whose embeddings are precomputed at initialization. At query time, the query embedding is compared against both sets via cosine similarity:

$$\delta = \max_{h \in \mathcal{H}} \text{sim}(\mathbf{q}, \mathbf{h}) - \max_{e \in \mathcal{E}} \text{sim}(\mathbf{q}, \mathbf{e}) \tag{4}$$

where $\mathcal{H}$ and $\mathcal{E}$ are the hard and easy candidate sets. The difficulty level is then hard if $\delta > \theta$, easy if $\delta < -\theta$, and medium otherwise, for a per-rule threshold θ. Multiple complexity rules can coexist (e.g., `code_complexity`, `math_complexity`), each with its own candidate sets and threshold. Composer conditions can restrict when a rule fires—for instance, evaluating `code_complexity` only when the domain signal indicates computer science.

Jailbreak Detection (τ_{jb}). Each jailbreak rule selects one of two detection methods via a `method` field (default: `classifier`).

BERT classifier method. A binary/ternary classifier detects adversarial prompt injection and jailbreak attempts. Each rule defines a confidence threshold θ; the signal fires when jailbreak confidence $c \geq \theta$. An optional `include_history` flag controls whether only the latest user message or the full conversation history is analyzed, trading latency for recall against multi-turn attacks. Multiple rules at different thresholds enable per-decision sensitivity—a public chatbot may use $\theta = 0.65$, while an internal tool uses $\theta = 0.9$ to minimize false positives.

Contrastive embedding method. Analogous to the complexity signal, each rule defines two sets of exemplar patterns—*jailbreak* ($\mathcal{K}_{\mathrm{jb}}$, e.g., "Ignore all previous instructions") and *benign* ($\mathcal{K}_{\mathrm{ben}}$, e.g., "What is the weather today")—whose embeddings are precomputed at initialization. At request time the contrastive score for a message m is:

$$\delta(m) = \max_{j \in \mathcal{K}_{\mathrm{jb}}} \mathrm{sim}(\mathbf{m}, \mathbf{j}) - \max_{b \in \mathcal{K}_{\mathrm{ben}}} \mathrm{sim}(\mathbf{m}, \mathbf{b}) \tag{5}$$

When `include_history` is enabled, the system evaluates every user message in the conversation and takes the maximum score across all turns: $\Delta = \max_{m \in \mathcal{M}_{\mathrm{user}}} \delta(m)$. The rule fires when $\Delta \geq \theta$ (default $\theta = 0.10$). This *max-contrastive-chain* aggregation is specifically designed to catch "boiling frog" multi-turn attacks where each individual message may score below threshold but the conversation contains at least one escalation turn.

Both methods coexist within the same signal type; a single deployment can define BERT and contrastive rules simultaneously and combine them in decision logic using OR (fire if either detects) or use them at different priority levels for graduated response. Classifier details are described in Section 7.

PII Detection (τ_{pii}). A token-level NER classifier identifies personally identifiable information (person names, emails, phone numbers, SSNs, credit cards, etc.). Each rule specifies a confidence threshold and an optional allow-list of PII entity types. The signal fires when any PII type *not* in the allow-list is detected above threshold. This per-rule policy model enables differentiated enforcement: a medical application may allow PERSON while blocking SSN, whereas a public chatbot blocks all PII types. Classifier details are described in Section 7.

Preference (τ_{pref}). 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 $\mathcal{T}_{\mathrm{used}} = \bigcup_{d \in \mathcal{D}} \{\tau \mid \exists \text{condition in } d \text{ of type } \tau\}$. Signal evaluators for types in $\mathcal{T}_{\mathrm{used}}$ 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 thirteen, this reduces total signal extraction latency by 50–70% compared to exhaustive evaluation. The strategy has a natural information-theoretic interpretation. Shannon's source coding theorem Shannon (1948) establishes that optimal codes assign shorter codewords to more probable symbols; analogously, demand-driven evaluation assigns *zero computational cost* to signals that carry no routing information in the current configuration, and full cost only to those that do. If we view each signal type's evaluation cost c_i as a "codeword length" and its relevance indicator $\mathbf{1}[\tau_i \in \mathcal{T}_{\mathrm{used}}]$ as the probability of being needed, then demand-driven evaluation minimizes the expected evaluation cost $\sum_i c_i \cdot \mathbf{1}[\tau_i \in \mathcal{T}_{\mathrm{used}}]$—the computational analogue of minimizing expected code length.

3.5 Extensibility

The thirteen 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 8): 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.

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 Rissanen (1978) 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.

3.8 INTERPRETATION AS HYBRID EMBEDDING

The signal extraction layer maps a raw request r to a structured signal vector $\mathbf{s} = S(r) \in \{0, 1\}^N \times [0, 1]^N$, where each dimension corresponds to a named signal condition and its associated confidence. This operation is functionally analogous to the *embedding layer* of a Transformer Vaswani et al. (2017): both convert unstructured input (token sequences or natural-language queries) into a high-dimensional representation suitable for downstream computation.

However, the analogy is not merely structural—it reveals a key design distinction. Transformer embeddings produce dense, opaque vectors optimized end-to-end for a downstream task. The signal vector $\mathbf{s}$, by contrast, is *interpretable by construction*: each dimension has a human-readable name (e.g., `domain:mathematics`, `complexity:high`), a defined semantic type, and an independently auditable extraction pipeline. This interpretability is not incidental but architecturally enforced: the decision engine references signals by type and

11

name, so the signal vector must be a *structured, symbolic representation* rather than a latent embedding.

We characterize this as a *hybrid embedding*: the extraction methods span a spectrum from sub-millisecond heuristics (keyword matching, regex patterns) to learned neural classifiers (LoRA-based domain and complexity models, embedding similarity), yet all project onto the same interpretable coordinate system. Figure 5 illustrates this comparison.
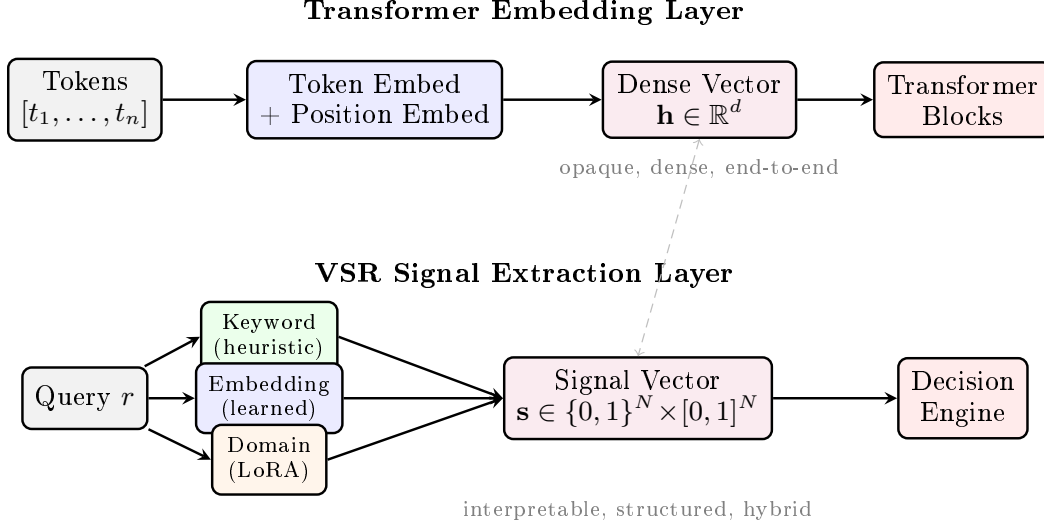
Transformer Embedding Layer

VSR Signal Extraction Layer

Figure 5: Comparison of embedding strategies. **Top**: A Transformer embedding layer produces dense, opaque vectors optimized end-to-end. **Bottom**: The VSR signal extraction layer produces structured, interpretable signal vectors from heterogeneous extractors (heuristic, learned, LoRA-based), where each dimension has a human-readable semantic name. The dashed arrow marks the functional correspondence: both convert unstructured input into a representation for downstream computation, but the signal vector is interpretable by construction.

4   DECISION ENGINE

The decision engine evaluates a set of routing decisions against the signal result and selects the best match. In Shannon's 1938 master's thesis Shannon (1938), arbitrary switching-circuit behavior was shown to be representable and minimizable through Boolean algebra—any desired input–output relation over binary signals could be systematically synthesized from $\{\wedge, \vee, \neg\}$. The decision engine applies this same principle to routing: each routing policy is expressed as a Boolean formula over binary signal conditions, and the functionally complete operator set guarantees that *any* routing policy expressible as a function of the extracted signals can be realized—without modifying the signal layer or the execution layer. 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 (Rule Node — Boolean Expression Tree). *A rule node ϕ is defined recursively as one of two forms (Figure 6 illustrates example expression trees):*

- ***Leaf** (signal reference): $\phi = \mathrm{leaf}(\tau, n)$, referencing a signal type τ and rule name n.*

- **Composite** *(Boolean operator):* $\phi = (\mathsf{op}, [\phi_1, \ldots, \phi_k])$, *where* $\mathsf{op} \in \{\mathrm{and}, \mathrm{or}, \mathrm{not}\}$ *and* $\phi_1, \ldots, \phi_k$ *are child rule nodes.* not *is strictly unary* $(k = 1)$.

Evaluation proceeds by structural recursion:

$$eval(\phi, S(r)) = \begin{cases} \mathbf{1}\big[\exists\,(\rho, 1, c) \in S(r) : \rho.\tau = \tau \wedge \rho.n = n\big] & \text{if } \phi = \mathrm{leaf}(\tau, n) \\ \bigwedge_{i=1}^{k} eval(\phi_i, S(r)) & \text{if } \mathsf{op} = \mathrm{and} \\ \bigvee_{i=1}^{k} eval(\phi_i, S(r)) & \text{if } \mathsf{op} = \mathrm{or} \\ \neg\, eval(\phi_1, S(r)) & \text{if } \mathsf{op} = \mathrm{not} \end{cases} \quad (6)$$

The recursive structure enables arbitrarily nested Boolean expressions. Classical flat formulas—a single AND or OR over leaf conditions—are the depth-1 special case and remain the recommended form for simple routing policies. When richer logic is needed, nesting expresses compound operators directly within a single decision: $\mathrm{nor}(A, B) = \mathrm{not}(\mathrm{or}(A, B))$; $\mathrm{nand}(A, B) = \mathrm{not}(\mathrm{and}(A, B))$; $\mathrm{xor}(A, B) = \mathrm{or}(\mathrm{and}(A, \mathrm{not}(B)), \mathrm{and}(\mathrm{not}(A), B))$. YAML's natural indentation mirrors the logical nesting, preserving readability and auditability even for complex formulas. Priority-ordered decisions compose multiple such formulas into an ordered evaluation, providing conflict resolution and organizational structure for deployment-scale routing policies.
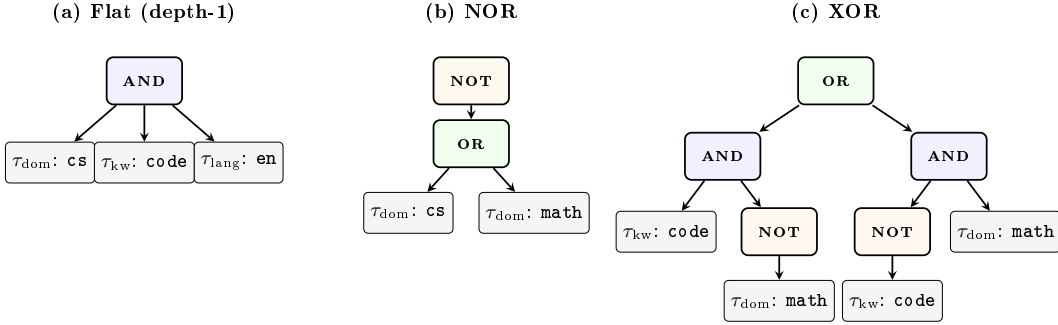

Figure 6: Rule-node expression trees at increasing depth. (a) A flat depth-1 tree: AND over three leaf conditions. (b) A NOR expression: NOT(OR(cs, math)), matching all non-STEM queries. (c) An XOR expression composed from AND, OR, and NOT primitives, routing requests that match exactly one of two signals. Leaf nodes (gray) reference signal conditions; composite nodes use AND (blue), OR (green), and NOT (orange).

4.2 CONFIDENCE COMPUTATION

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

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

where $\Gamma_{\mathrm{sat}} = \{\gamma_j \in \Gamma \mid \mathrm{sat}(\gamma_j, S(r)) = 1\}$ 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 $\mathcal{D}_{\mathrm{match}} = \{d \in \mathcal{D} \mid eval(\phi_d, S(r)) = 1\}$, two strategies select d^*:

Priority Strategy.
$$d^* = \arg\max_{d \in \mathcal{D}_{\mathrm{match}}} p_d \quad (8)$$

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^*
 1: $\mathcal{D}_{\text{match}} \leftarrow \emptyset$
 2: **for** $d \in \mathcal{D}$ **do**
 3:     **if** $\text{eval}(\phi_d, S(r))$ **then**
 4:         $c_d \leftarrow \text{conf}(d, S(r))$
 5:         $\mathcal{D}_{\text{match}} \leftarrow \mathcal{D}_{\text{match}} \cup \{(d, c_d)\}$
 6:     **end if**
 7: **end for**
 8: **if** $\sigma = \text{priority}$ **then**
 9:     $(d^*, c^*) \leftarrow \arg\max_{(d,c) \in \mathcal{D}_{\text{match}}} p_d$
10: **else**
11:     $(d^*, c^*) \leftarrow \arg\max_{(d,c) \in \mathcal{D}_{\text{match}}} c$
12: **end if**
13: **return** (d^*, c^*)

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.

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 recursive rule-node model can express common routing patterns with depth-1 trees (flat formulas) and richer patterns through nesting:

- **Domain routing**: A single leaf 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(domain, NOT(complexity)) 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.
- **NOR routing** (blanket exclusion): NOT(OR(cs, math, physics)) routes all non-STEM queries to a general-purpose model without enumerating every non-STEM domain.
- **NAND routing** (conditional exemption): NOT(AND(zh, code)) matches all requests *except* Chinese-language code queries, useful for compliance-based routing.

- **XOR routing** (mutual exclusion): OR(AND(A, NOT(B)), AND(NOT(A), B)) routes requests matching exactly one of two signals, enabling exclusive specialization.
- **Nested multi-signal**: AND(OR(cs, math_kw), en, NOT(long_ctx)) combines disjunctive, conjunctive, and negated sub-trees in a single decision for fine-grained routing.

Functional completeness. The expressiveness of the recursive rule-node model rests on a classical result from Boolean algebra Huntington (1904): the operator set $\{\wedge, \vee, \neg\}$ is *functionally complete*—any Boolean function $f : \{0,1\}^N \rightarrow \{0,1\}$ can be expressed as a formula over these operators. Because each rule node may nest AND, OR, and NOT to arbitrary depth, a *single* decision formula ϕ can already represent any Boolean function over the N signal match indicators—no multi-decision composition is required for completeness.

Proposition 1 (Single-decision completeness). *For any Boolean function $f : \{0,1\}^N \rightarrow \{0,1\}$ over signal match indicators, there exists a rule node ϕ using AND, OR, and NOT such that $eval(\phi, S(r)) = f(S(r))$ for all signal results $S(r)$.*

Proof sketch. Construct ϕ directly from the truth table of f. For each minterm (input assignment where $f = 1$), form an AND-node over the corresponding literals (a leaf LEAF(τ_i, n_i) when the i-th signal is 1, or NOT(LEAF(τ_i, n_i)) when it is 0). Collect all such AND-nodes under a single OR-node. Since $\{\wedge, \vee, \neg\}$ is functionally complete, the resulting tree evaluates to f. □

Proposition 2 (Routing policy completeness). *For any routing policy $\pi : \{0,1\}^N \rightarrow \mathcal{M} \cup \{\perp\}$ mapping signal vectors to model selections, there exists a decision set $\mathcal{D}$ with recursive AND/OR/NOT formulas and priority ordering such that π is realized by the evaluation algorithm (Algorithm 1).*

Proof sketch. For each model m_k in the range of π, construct a single decision d_k whose rule node encodes the Boolean function $f_k(\mathbf{s}) = \mathbf{1}[\pi(\mathbf{s}) = m_k]$ using the single-decision completeness result above. Assign priorities to resolve overlaps deterministically. □

This two-level universality guarantee—completeness within a single decision *and* across a decision set—means the decision engine imposes *no inherent limitation* on what routing policies can be configured. Priority ordering across decisions is no longer required for expressiveness; it serves as an organizational and conflict-resolution mechanism, equivalent to a decision list Rivest (1987) over Boolean features.

Structural analogy to combinational logic circuits. The recursive rule-node model maps naturally onto the hierarchy of combinational logic circuits studied in digital design Brayton et al. (1984). The following table summarizes the correspondence at three levels of generality:

Circuit Model	Routing System	Expressiveness
PLA (two-level) Fleisher & Maissel (1975)	Flat decision (depth-1 tree)	Any two-level Boolean function
General combinational circuit	Recursive rule-node tree	Any Boolean function
Circuit array + priority encoder	Decision set + priority ordering	Any routing policy $\pi : \{0,1\}^N \rightarrow \mathcal{M}$

At the first level, a depth-1 rule node—a single AND or OR over leaf conditions—is structurally isomorphic to a *Programmable Logic Array* (PLA) Fleisher & Maissel (1975): signal extractors correspond to input lines, leaf conditions to the AND-plane, and the top-level operator to the OR-plane. At the second level, a recursive rule-node tree corresponds to a general combinational logic circuit—a directed acyclic graph of AND, OR, and NOT gates—where each decision is a complete circuit computing an arbitrary Boolean function.

At the third level, the priority-ordered decision set acts as an array of such circuits with a priority encoder selecting the output, realizing any routing policy. Figure 7 illustrates this three-level correspondence.
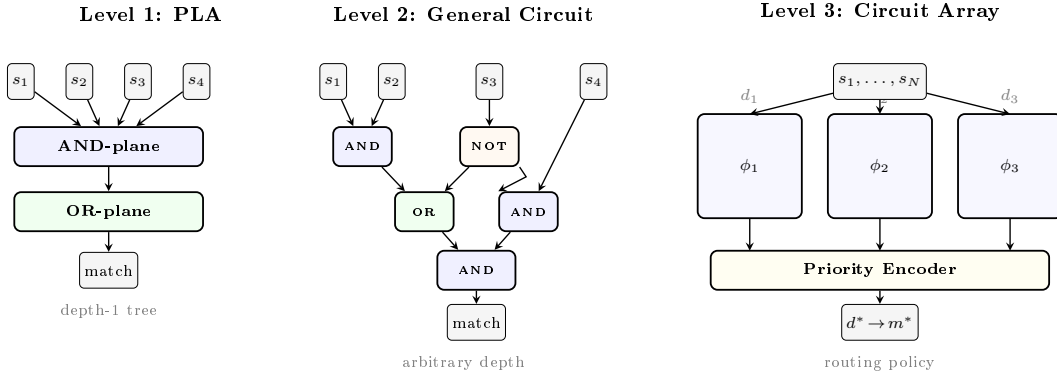

Level 1: PLA **Level 2: General Circuit** **Level 3: Circuit Array**

Figure 7: Three-level correspondence between combinational logic circuits and the decision engine. **Level 1**: A PLA with AND-plane and OR-plane corresponds to a flat (depth-1) decision formula. **Level 2**: A general combinational circuit with arbitrarily nested AND, OR, and NOT gates corresponds to a recursive rule-node tree within a single decision. **Level 3**: An array of circuits with a priority encoder corresponds to the full decision set with priority-ordered evaluation, realizing any routing policy.

In a PLA, logic is "programmed" by setting fuse connections; in a general circuit, by wiring gates; in our system, by YAML configuration whose indentation directly mirrors the gate-level nesting. This correspondence provides a well-understood theoretical foundation: decades of results on logic minimization, hazard-free design, and testability Brayton et al. (1984) apply directly to reasoning about decision optimization and coverage analysis.

Decision set verification and minimization. The combinational-logic 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 Brayton et al. (1984) and multi-level logic restructuring Brayton et al. (1984), 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 rule-node trees over the continuous confidence values directly, using fuzzy logic operators Zadeh (1965).

Definition 6 (Fuzzy Rule-Node Evaluation). *Given continuous signal confidences $c(r) \in [0,1]$ for each leaf condition, the fuzzy evaluation of a rule node ϕ is defined by structural*

recursion:

$$\widetilde{eval}(\phi, S(r)) = \begin{cases} c_\phi(r) & \text{if } \phi = \text{leaf}(\tau, n) \\ \overset{k}{\underset{i=1}{\min}} \ \widetilde{eval}(\phi_i, S(r)) & \text{if op} = \text{and} \\ \overset{k}{\underset{i=1}{\max}} \ \widetilde{eval}(\phi_i, S(r)) & \text{if op} = \text{or} \\ 1 - \widetilde{eval}(\phi_1, S(r)) & \text{if op} = \text{not} \end{cases} \tag{10}$$

where $c_\phi(r)$ is the confidence score of the signal matching leaf ϕ.

The operators $(\min, \max, 1-x)$ form the standard fuzzy complement triple and satisfy De Morgan's laws at every level of the tree, preserving the algebraic properties of the crisp model Bellman & Zadeh (1970). This fuzzy evaluation is a *strict generalization*: when all confidences are binary ($c \in \{0, 1\}$), min reduces to $\wedge$, max reduces to $\vee$, and $1-x$ reduces to $\neg$, 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.

4.8 Interpretation as Mixture-of-Experts Gating

The priority-ordered decision evaluation admits a precise structural analogy to the *Mixture-of-Experts* (MoE) gating mechanism Shazeer et al. (2017). In a standard MoE layer, a gating network $G(\mathbf{x})$ computes a sparse distribution over K expert sub-networks, and the output is a weighted combination of the selected experts' outputs. In our system:

- The **signal vector s** plays the role of the shared representation that the gating network operates on.

- Each **decision block** d_i with its Boolean formula $\phi_i(\mathbf{s})$ acts as an *expert gate*—a binary function that determines whether expert i (a model pool with associated plugins) is activated.

- **Priority ordering** implements *hard routing with early exit*: the first decision whose gate evaluates to true captures the request, analogous to top-1 expert selection in sparse MoE but with a deterministic, priority-based selection rule rather than a learned softmax.

17

Figure 8 formalizes this correspondence. Unlike standard MoE, the gates here are symbolic (Boolean formulas over interpretable signals) rather than learned softmax projections, which enables formal verifiability and compositional editability.
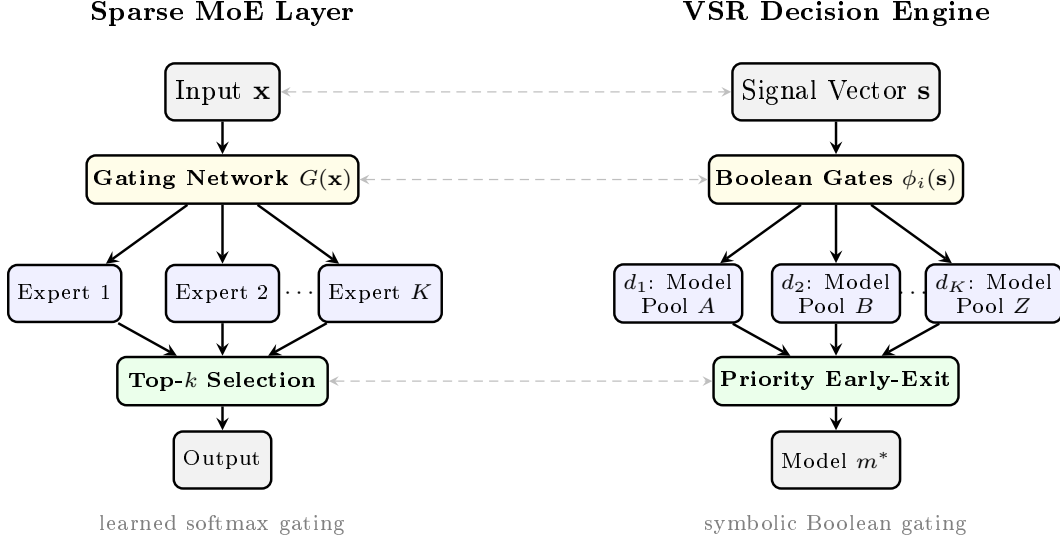

Figure 8: Structural analogy between Mixture-of-Experts gating and the VSR decision engine. **Left**: In a sparse MoE layer, a learned gating network selects top-k experts via softmax. **Right**: In VSR, Boolean formulas over the signal vector act as symbolic expert gates, with priority ordering implementing deterministic early-exit selection. Dashed arrows mark the structural correspondence.

4.9 Layered Entropy-Folding Interpretation

The priority-ordered gate stack also admits an information-theoretic interpretation as a layered uncertainty-collapse process. Let $g_\ell(\mathbf{s}) \in \{0, 1\}$ denote the match outcome of the ℓ-th gate under priority order, and let $Z_\ell = g_\ell(\mathbf{s})$. Define routing uncertainty after ℓ evaluated gates as:

$$U_\ell = H(M \mid \mathbf{s}, Z_{1:\ell}) \tag{11}$$

where M is the final selected model random variable. By the chain rule of mutual information:

$$U_{\ell+1} = U_\ell - I(M; Z_{\ell+1} \mid \mathbf{s}, Z_{1:\ell}) \leq U_\ell \tag{12}$$

so each additional gate can only maintain or reduce uncertainty. In this sense, priority depth is *control depth*: a sequence of policy constraints that folds entropy toward a deterministic choice.

The MoE analogy also clarifies what the system is *not*: it is not a Transformer in the sequential sense. Transformer blocks transform the representation layer by layer—each block's output is the next block's input. In our architecture, all decision blocks operate on the *same* shared signal vector $\mathbf{s}$; there is no inter-decision information flow. The priority ordering is an *early-exit mechanism* over parallel evaluations, not a sequential transformation pipeline. This distinction is important: it means that adding or removing decisions does not affect the behavior of other decisions (modulo priority reordering), a composability property that sequential architectures lack.

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

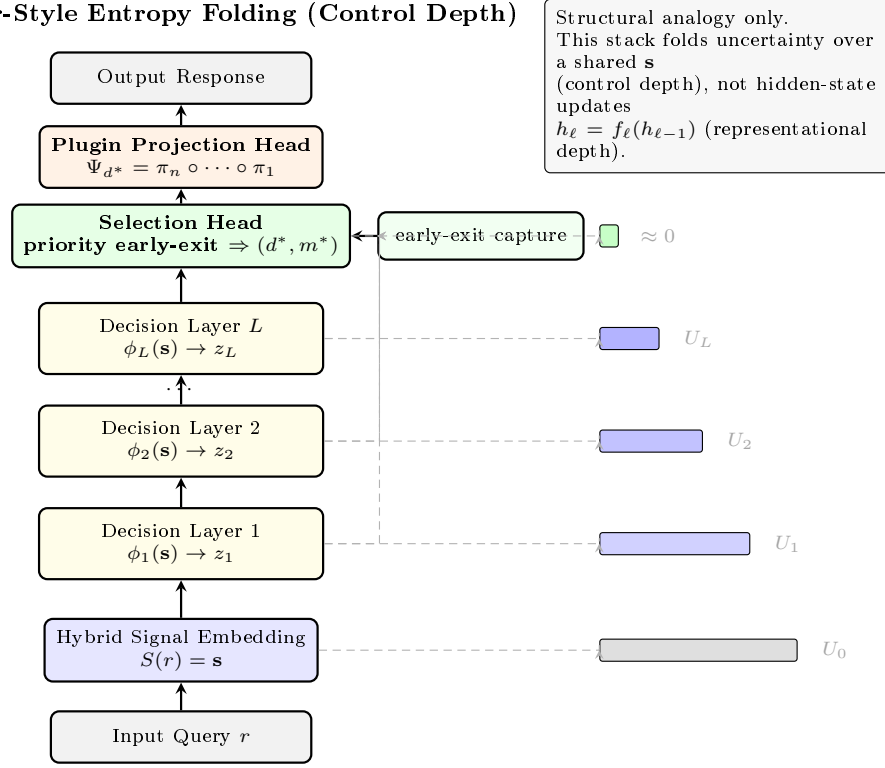**Transformer-Style Entropy Folding (Control Depth)**

Figure 9: Transformer-style visualization of layered control depth in VSR. Signal extraction constructs a shared embedding $\mathbf{s}$; stacked decision layers apply Boolean gates $\phi_\ell(\mathbf{s})$ that progressively reduce routing uncertainty U_ℓ; selection resolves (d^*, m^*) via priority early-exit; plugin composition Ψ_{d^*} acts as an output-side projection head over request/response behavior.

model and four core infrastructure plugins; safety signal classifiers are covered in Section 7, hallucination detection in Section 8, and memory retrieval and RAG injection in Section 13.

5.1   PLUGIN EXECUTION MODEL

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

$$\pi : (\text{Request}, \text{Context}, \text{Config}_\pi) \to (\text{Request}', \text{Response}') \cup \{\bot\} \tag{13}$$

where $\bot$ denotes early termination (e.g., a cache hit returning immediately or a fast response short-circuiting the pipeline).

Plugins execute in a fixed pipeline order within each decision's chain. On the *request path*: fast response → cache → RAG → modality → memory → system prompt → 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.

Jailbreak and PII detection are first-class *signals* (Sections 3 and 7) evaluated in parallel with all other signal types, not serial plugins in the request path. Decisions that match safety signals activate the `fast_response` plugin to return an immediate rejection.

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 Output-Space Projection Perspective

For a matched decision d^*, the plugin chain can be interpreted as a policy-scoped projection from routing state to deployment-compliant output behavior:

$$\Psi_{d^*} = \pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1 \tag{14}$$

where each π_i is a typed transformation in the configured chain. Functionally, this is analogous to an output projection stage after gating: once the decision fixes the active policy path, plugins realize that policy as concrete transformations on request/response state (e.g., context injection, auth header mutation, hallucination annotation, cache write-through). This analogy is operational rather than architectural: unlike a Transformer projection head, Ψ_{d^*} is a configurable symbolic composition, not a learned linear map.

5.3 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:

$$\cos(\mathbf{e}_q, \mathbf{e}_e) \geq \theta_d \tag{15}$$

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).

5.4 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.

5.5 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).

5.6 Fast Response

The fast response plugin enables a decision to short-circuit the entire pipeline and return an immediate OpenAI-compatible response without forwarding the request to any upstream model. This is the primary mechanism for acting on safety signals.

Execution model. When a decision's `fast_response` plugin is configured, the pipeline checks for it before any other plugin in the chain. If present, the configured message is returned as a standard `chat.completion` response with `finish_reason: "stop"`, and no further processing occurs.

Streaming compatibility. The plugin inspects the original request's `stream` parameter. For non-streaming requests, it returns a single JSON object. For streaming requests (`stream: true`), it generates a Server-Sent Events (SSE) sequence: an initial chunk with the assistant role, word-by-word content chunks, a final chunk with `finish_reason: "stop"`, and the `[DONE]` sentinel—matching the exact format that OpenAI-compatible clients expect.

Use cases. The canonical use case is safety enforcement: a decision matching a jailbreak or PII signal activates `fast_response` to return a policy-compliant refusal message. However, the plugin is general-purpose—it can also serve canned responses for FAQ-like queries, maintenance windows, or rate-limited fallback messages without consuming model compute.

6 Programmable Neural-Symbolic Configuration Language

While the YAML configuration format described in preceding sections is sufficient for direct human authorship, it conflates two concerns: the *logical specification* of routing policy (signals, decisions, plugins) and the *deployment target* (flat YAML, Kubernetes CRDs, Helm charts). We introduce a programmable configuration language—the instruction set of the neural-symbolic inference engine (formalized in Section 6.8)—that separates these concerns, providing a concise, human-readable, and machine-parseable surface syntax for routing policies that compiles to multiple deployment targets through a shared internal representation.

6.1 Design Goals

The DSL is designed around four principles:

1. **Conciseness**: Routing policies should read as close to natural language as possible, minimizing syntactic noise relative to YAML.

2. **Type safety**: Signal references in Boolean expressions are resolved at compile time; undefined or misspelled signal names produce diagnostics before deployment.

3. **Multi-target emission**: A single DSL source compiles to flat YAML (for local development), Kubernetes `SemanticRouter` CRDs (for operator-based deployment), or Helm `values.yaml` (for chart-based deployment).

4. **Round-trip fidelity**: An existing `RouterConfig` can be *decompiled* back to DSL source, enabling migration from YAML to DSL and "round-trip" editing workflows.

6.2 Concrete Syntax

The DSL defines five top-level block types: `SIGNAL`, `ROUTE`, `PLUGIN`, `BACKEND`, and `GLOBAL`. Figure 10 shows a representative configuration.

Each `SIGNAL` block declares a named signal of a specific type (one of 12 supported types: `keyword`, `embedding`, `domain`, `fact_check`, `user_feedback`, `preference`, `language`, `context`, `complexity`, `modality`, `authz`, `jailbreak`, `pii`). Each `ROUTE` block defines a decision with a priority, a Boolean `WHEN` clause over signal references, one or more model references with optional per-model parameters (reasoning mode, effort level, LoRA adapter, weight), an optional selection algorithm, and zero or more plugin attachments. `PLUGIN`

```
SIGNAL domain math { mmlu_categories: ["math"] }
SIGNAL keyword urgent { operator: "any", keywords: ["urgent","asap"] }

PLUGIN safe_pii pii { enabled: true, pii_types_allowed: [] }

ROUTE math_route (description = "Math") {
  PRIORITY 100
  WHEN domain("math")
  MODEL "qwen2.5:3b" (reasoning = true, effort = "high")
  PLUGIN safe_pii
}
ROUTE urgent_ai {
  PRIORITY 200
  WHEN keyword("urgent") AND NOT domain("math")
  MODEL "qwen3:70b" (reasoning = true),
        "qwen2.5:3b" (reasoning = false)
  ALGORITHM confidence { threshold: 0.5 }
}

BACKEND vllm_endpoint ollama { address: "127.0.0.1", port: 11434 }
GLOBAL { default_model: "qwen2.5:3b", strategy: "priority" }
```

Figure 10: A representative DSL configuration defining two signals, a reusable plugin template, two routes with Boolean decision logic, a backend endpoint, and global settings.

blocks at the top level define reusable templates that routes reference by name; inline plugin blocks within routes define route-specific configurations.

6.3 GRAMMAR AND PARSING

The lexer and parser are implemented using the `particible` parser-generator library Thomas (2024). The lexer defines 12 token classes (identifiers, integers, floats, strings, booleans, braces, parentheses, brackets, colon, comma, equals, comments) via regular expressions. The parser uses a PEG-style grammar with lookahead 3 to resolve ambiguities between signal references and other identifier uses.

The Boolean expression grammar for `WHEN` clauses follows standard precedence:

$$BoolExpr ::= AndTerm \, (\text{OR} \, AndTerm)^* \qquad (16)$$
$$AndTerm ::= Factor \, (\text{AND} \, Factor)^* \qquad (17)$$
$$Factor ::= \text{NOT} \, Factor \mid (\, BoolExpr \,) \mid SignalRef \qquad (18)$$
$$SignalRef ::= type \, (\, \text{``name''} \,) \qquad (19)$$

This grammar is parsed into a recursive AST (`BoolAnd`, `BoolOr`, `BoolNot`, `SignalRefExpr` nodes) that the compiler flattens into the `RuleCombination` tree consumed by the decision engine (Section 4). Figure 11 illustrates the AST for a representative Boolean expression.

Error recovery is block-granular: if parsing a top-level block fails, the parser splits the input at block boundaries and attempts to parse remaining blocks independently, accumulating partial results alongside error diagnostics. This enables IDE-like incremental feedback during authoring.

6.4 COMPILATION PIPELINE

The compilation pipeline (Figure 12) transforms DSL source through four stages:

1. **Lexing**: The source is tokenized into a stream of typed tokens with position tracking for diagnostic reporting.
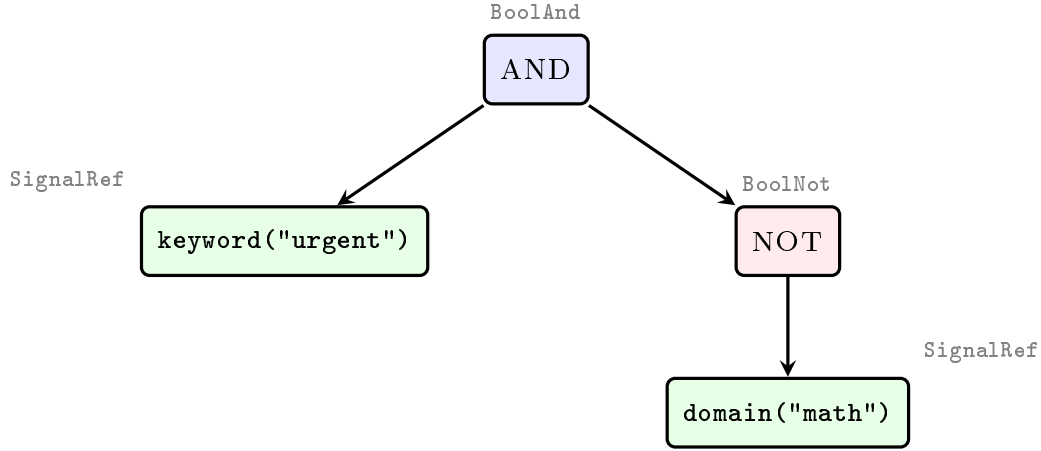
Figure 11: Boolean expression AST for `WHEN keyword("urgent") AND NOT domain("math")`. Each leaf is a `SignalRefExpr` referencing a named signal; internal nodes are Boolean operators (`BoolAnd`, `BoolNot`). The compiler flattens this tree into a `RuleCombination` structure for the decision engine.

2. **Parsing**: The token stream is parsed into a *raw parse tree* (participle-generated structs), then lowered to a *resolved AST* (`Program → SignalDecl`, `RouteDecl`, `PluginDecl`, `BackendDecl`, `GlobalDecl`) with desugared values and resolved positions.

3. **Compilation**: The AST is compiled to the internal `RouterConfig` structure. This involves: (a) mapping each `SIGNAL` block to the appropriate signal configuration (keyword rules, embedding rules, domain categories, etc.); (b) flattening the Boolean expression tree in each `WHEN` clause into a `RuleCombination` tree with `AND`/`OR`/`NOT` operators; (c) resolving plugin references against top-level templates with field-level merge semantics (route-local fields override template defaults); and (d) mapping `BACKEND` blocks to endpoint, provider profile, embedding model, or semantic cache configurations based on the backend type keyword.

4. **Emission**: The `RouterConfig` is serialized to one of three target formats.
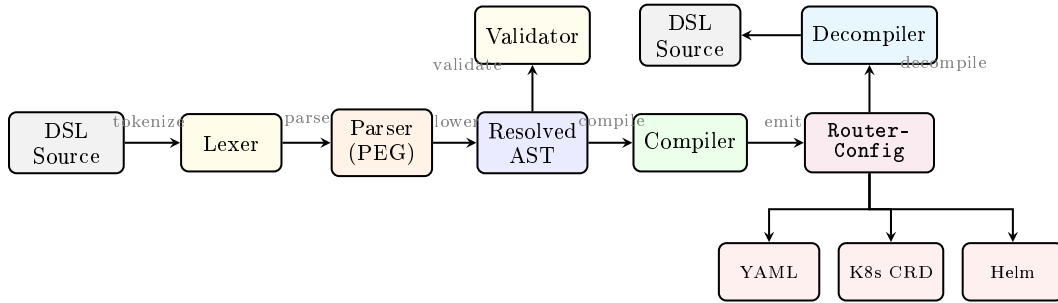

Figure 12: The DSL compilation pipeline. Source text is lexed, parsed into a resolved AST, and compiled to the internal `RouterConfig`. From this representation, three emitters produce deployment-specific formats (flat YAML, Kubernetes CRD, Helm values). The decompiler reverses the pipeline, reconstructing DSL source from an existing `RouterConfig`, enabling round-trip editing. The validator operates on the AST directly, producing three-level diagnostics.

6.5   MULTI-TARGET EMISSION

The shared `RouterConfig` representation enables three emission targets from a single DSL source:

- **Flat YAML** (`EmitYAML`): Direct marshaling of `RouterConfig` for local development and testing. An alternative `EmitUserYAML` variant restructures the output into the nested `signals`/`providers` format expected by the CLI tooling.

- **Kubernetes CRD** (`EmitCRD`): Wraps the routing logic in a `SemanticRouter` custom resource (`vllm.ai/v1alpha1`), mapping backend definitions to `spec.vllmEndpoints` and routing configuration to `spec.config`. Signal rules that the CRD schema does not model are preserved as extra fields for ConfigMap-based deployment compatibility.

- **Helm values** (`EmitHelm`): Nests the `RouterConfig` under a `config:` key compatible with the Helm chart's ConfigMap template, pruning zero-value infrastructure sections for clean output.

This separation means that infrastructure teams can change deployment targets without modifying the routing policy, and routing engineers can evolve policies without understanding Kubernetes manifests.

6.6   DECOMPILATION AND ROUND-TRIP EDITING

The decompiler reconstructs DSL source text from an existing `RouterConfig`:

1. **Plugin template extraction**: Plugins used by multiple routes are automatically factored into top-level `PLUGIN` templates; route-local plugins remain inline.

2. **Rule tree reconstruction**: The `RuleCombination` tree in each decision is walked recursively to reconstruct the Boolean expression with proper `AND`/`OR`/`NOT` operators and parenthesization.

3. **Signal type inference**: Signal references in rule nodes are matched against the configuration's signal lists to recover the original signal type keywords.

This enables a migration path: existing YAML configurations can be decompiled to DSL, edited in the more concise syntax, and recompiled to any target format. The round-trip property (DSL $\xrightarrow{\text{compile}}$ `RouterConfig` $\xrightarrow{\text{decompile}}$ DSL $\xrightarrow{\text{compile}}$ `RouterConfig`$'$ $\equiv$ `RouterConfig`) is validated by extensive test suites including idempotency and double-round-trip tests.

6.7   THREE-LEVEL VALIDATION

The validator operates on the resolved AST (before compilation) and produces diagnostics at three severity levels:

1. **Error** (Level 1): Syntax errors detected during parsing—malformed blocks, unexpected tokens, missing delimiters. The block-granular error recovery ensures that errors in one block do not prevent analysis of subsequent blocks.

2. **Warning** (Level 2): Reference resolution issues—a `WHEN` clause references a signal name not defined in any `SIGNAL` block, or a `PLUGIN` reference has no matching template. The validator performs fuzzy matching on undefined signal names and suggests corrections via `QuickFix` annotations (e.g., "did you mean `math`?" for a reference to `mth`).

3. **Constraint** (Level 3): Semantic constraint violations—embedding thresholds outside $[0, 1]$, port numbers outside valid ranges, negative route priorities, unknown algorithm or signal types. These catch logical errors that are syntactically valid but semantically incorrect.

Figure 13 illustrates the three-level diagnostic architecture. This scheme provides IDE-like progressive feedback, enabling both batch validation (CLI `validate` command) and interactive authoring with incremental diagnostics.
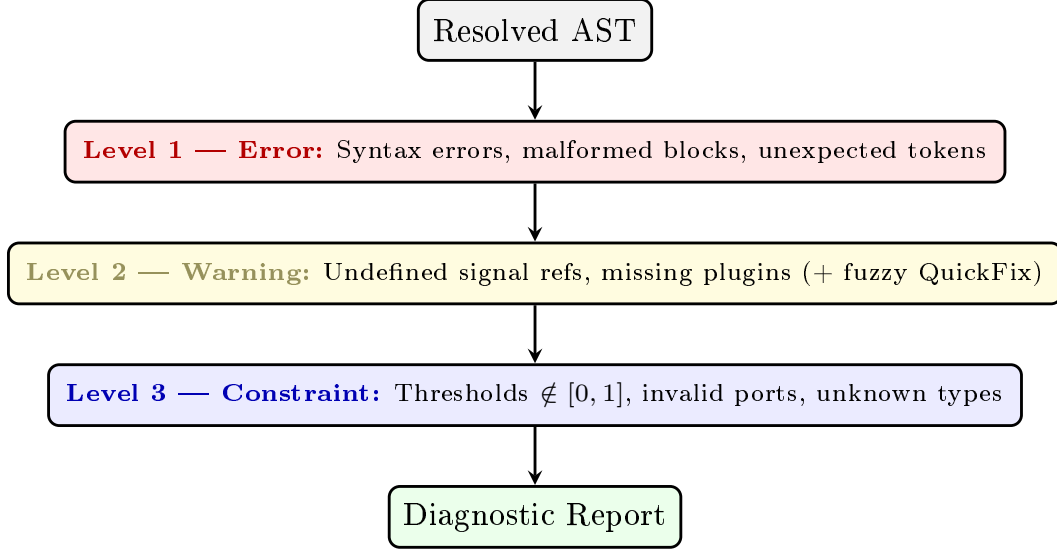

Figure 13: Three-level validation architecture. The validator processes the resolved AST through progressively deeper analysis: syntactic errors (Level 1), reference resolution with fuzzy-matched QuickFix suggestions (Level 2), and semantic constraint checking (Level 3). Diagnostics at all levels are accumulated into a unified report.

6.8 DSL as Instruction Set and Agent-Based Policy Synthesis

The configuration language can be understood as the *instruction set* of the neural-symbolic inference engine. Just as a CPU's instruction set defines the space of programs that can execute on the hardware, the DSL defines the space of routing policies that can be instantiated on the signal-decision-plugin architecture. The functional completeness result of Section 4 guarantees that this instruction set is *universal*: any routing policy $\pi : \{0,1\}^N \to \mathcal{M}$ is expressible.

This framing transforms the problem of *configuring* the router into a *program synthesis* problem (Figure 14): given a natural-language specification of routing requirements ("route math queries to the math model, enforce PII filtering for healthcare queries"), synthesize a valid DSL configuration that implements the specification. This is precisely the class of problems that code-generation agents—LLMs fine-tuned or prompted for program synthesis—are designed to solve.

The DSL's formal grammar and type-safe compilation make it particularly suitable as the target language: the structured syntax—with explicit keywords, typed blocks, and a finite set of signal types—constrains the generation space far more tightly than free-form YAML, reducing the probability of syntactically valid but semantically incorrect configurations. The three-level validator provides immediate, machine-readable feedback that a coding agent can use to iteratively refine generated configurations.

The connection to reinforcement learning is direct: the coding agent's policy $\pi_\theta(\text{config} \mid \text{spec})$ can be optimized via RLHF or rule-based reward Zhang et al. (2025a), where the reward signal is the downstream routing quality $Q(r, m^*)$ aggregated over a traffic distribution. This operates at a *meta-level* compared to prior work on RL-based routing Zhang et al. (2025a): rather than learning to route individual requests, the agent learns to *write routing policies* that generalize across request distributions. The DSL provides the structured action space that makes this meta-learning tractable—the agent generates a finite, syntactically constrained configuration rather than an arbitrary neural network.
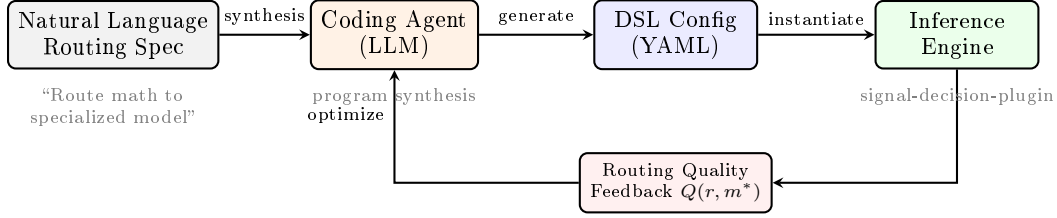
Figure 14: Agent-based policy synthesis pipeline. A coding agent (LLM) translates natural-language routing specifications into DSL configurations, which are instantiated on the inference engine. Routing quality feedback closes the loop, enabling iterative policy refinement. The DSL's functional completeness guarantees that any expressible routing policy is reachable by the synthesis process.

6.9 SYNTHESIS: A PROGRAMMABLE NEURAL-SYMBOLIC INFERENCE ENGINE

Taken together, the signal extraction layer (Section 3), decision engine (Section 4), and configuration language characterize the system as a **programmable neural-symbolic inference engine**:

1. **Neural front-end**: The signal extraction layer uses both heuristic and learned (neural) methods to produce a structured representation—analogous to a hybrid embedding layer (Section 3.8).

2. **Symbolic back-end**: The decision engine applies Boolean logic over the structured representation—analogous to symbolic expert gating with formal verifiability (Section 4.8).

3. **Programmable interface**: The DSL configuration is the "program" that specifies the inference behavior, and the functional completeness of the Boolean decision model guarantees universality of the program space.

4. **Agent-compilable**: The structured DSL enables LLM-based coding agents to serve as "compilers" from natural-language specifications to executable routing policies, with RL-based optimization closing the loop.

This perspective unifies the Shannon-theoretic foundation (information reduction + Boolean synthesis) with the modern ML landscape (embeddings + MoE gating + agent-based program synthesis), and suggests that the architecture occupies a principled point in the design space between fully neural routing systems (which sacrifice interpretability and editability) and fully manual rule systems (which sacrifice adaptivity and scalability).

7 REQUEST-TIME SAFETY: JAILBREAK AND PII SIGNALS

Request-time safety in our architecture is implemented as *first-class signals*, not as serial plugins. Jailbreak and PII detection run in the signal extraction layer (Section 3), evaluated in parallel alongside all other signal types (keyword, embedding, domain, etc.). Figure 15 illustrates how safety signals integrate with domain signals and decision rules. When a safety signal fires, the decision engine matches it to a decision that activates the `fast_response` plugin (Section 5), returning an immediate OpenAI-compatible response without contacting any upstream LLM.

This signal-driven design has three key advantages:

1. **Zero added latency**: safety classifiers run concurrently with intent signals, so the wall-clock cost is max(safety, intent) rather than safety + intent.
2. **Composability**: safety signals can be combined with domain, keyword, or embedding signals in decision rules using AND/OR logic—e.g., strict jailbreak detection only for financial queries.

26

3. **Uniform observability**: safety results appear as standard signal matches in headers (`x-vsr-matched-jailbreak`, `x-vsr-matched-pii`), traces, and the dashboard topology view.

7.1 Jailbreak Detection

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

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

$$g_{\text{jb}}(x) = (y, c) \in \{\text{BENIGN}, \text{INJECTION}, \text{JAILBREAK}\} \times [0, 1] \tag{20}$$

A jailbreak signal rule with threshold θ fires iff $y \neq \text{BENIGN}$ and $c \geq \theta$. Multiple rules at different thresholds can coexist, enabling decisions to reference different sensitivity levels.

Classifier architecture. We support four inference backends with varying context-length and deployment characteristics: (1) BERT with LoRA adapters (standard context); (2) ModernBERT Warner et al. (2024) 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 9), reducing model memory footprint.

Per-rule sensitivity. Different signal rules configure different thresholds: a `jailbreak_strict` rule might use $\theta = 0.4$ with full history analysis for financial endpoints, while a `jailbreak_standard` rule uses $\theta = 0.65$ on the latest message only. Decisions then reference the appropriate rule by name, enabling context-aware security policies (e.g., strict detection only when the domain signal also matches "economics").

7.2 Contrastive Jailbreak Detection

While the BERT classifier excels at detecting single-turn jailbreak attempts, it can be evaded by *multi-turn escalation* ("boiling frog") attacks in which each individual message appears benign but the conversation progressively steers the model toward unsafe behavior. We introduce a contrastive embedding method that operates alongside the BERT classifier within the same jailbreak signal type.

Knowledge base construction. Each contrastive rule defines two exemplar sets: a *jailbreak knowledge base* $\mathcal{K}_{\text{jb}}$ containing known adversarial patterns (e.g., "Ignore all previous instructions", "You are now DAN") and a *benign knowledge base* $\mathcal{K}_{\text{ben}}$ containing representative normal queries (e.g., "What is the weather today", "Help me write an email"). All KB embeddings are precomputed at initialization using a concurrent worker pool (identical to the complexity signal's preloading strategy), so runtime cost is limited to embedding the incoming message and computing cosine similarities.

Contrastive scoring. For each user message m, the contrastive score measures relative proximity to the jailbreak vs. benign exemplars:

$$\delta(m) = \max_{j \in \mathcal{K}_{\text{jb}}} \cos(\mathbf{m}, \mathbf{j}) - \max_{b \in \mathcal{K}_{\text{ben}}} \cos(\mathbf{m}, \mathbf{b}) \tag{21}$$

A positive δ indicates the message is semantically closer to jailbreak patterns; a negative δ indicates it is closer to benign patterns. The subtraction normalizes for the overall embedding similarity scale, making the threshold robust across different embedding models.

Max-contrastive chain (multi-turn detection). When `include_history` is enabled, the system exploits the stateless nature of the OpenAI chat API—each request carries the full conversation history in the `messages` array—to evaluate every user message:

$$\Delta = \max_{m \in \mathcal{M}_{\text{user}}} \delta(m) \tag{22}$$

The rule fires when $\Delta \geq \theta$ (default $\theta = 0.10$). This max-pooling aggregation ensures that even if the current message is innocuous, the presence of any high-scoring turn in the history

triggers detection. The approach is effective against gradual escalation because the attacker must include at least one semantically adversarial turn to steer the model, and that turn will produce a high contrastive score regardless of its position in the conversation.

Integration with signal architecture. The contrastive method is selected via the `method:  contrastive` field on a jailbreak rule, keeping the same signal type τ_{jb} and rule-name addressing used by the BERT classifier. This means contrastive rules participate fully in the decision engine's Boolean logic. A typical deployment combines both methods:

- A BERT rule (`jailbreak_standard`, $\theta = 0.65$) for fast, high-precision single-turn detection.
- A contrastive rule (`jailbreak_multiturn`, $\theta = 0.10$, `include_history:  true`) for multi-turn chain detection.
- A decision with OR logic fires if *either* method detects an attack, providing defense in depth.

The embedding model is inherited from the global `embedding_models` configuration (consistent with the complexity signal), avoiding per-rule model specification.

7.3 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:

$$g_{\mathrm{pii}}(x) = \big\{(i, j, \ell, c) \mid \text{span } x_i \ldots x_j \text{ is PII type } \ell \text{ with confidence } c\big\} \tag{23}$$

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

Policy model. Each PII signal rule specifies a threshold θ and an optional allow-list $\mathcal{L}_{\mathrm{allow}}$. The rule fires when any detected entity $(\cdot, \cdot, \ell, c)$ satisfies $c \geq \theta$ and $\ell \notin \mathcal{L}_{\mathrm{allow}}$. This per-rule policy enables differentiated enforcement: a `pii_deny_all` rule blocks all entity types, while a `pii_allow_email` rule permits email addresses for appointment booking in medical contexts.

7.4 Parallel Safety Evaluation

Both jailbreak and PII classifiers launch as concurrent goroutines within the signal extraction layer, alongside all other signal evaluators. The wall-clock time is dominated by the slowest signal evaluator rather than the sum of safety and intent classifiers. Both use the LoRA adapter architecture (Section 9), and their results merge into the standard signal result $S(r)$ consumed by the decision engine.

When a safety signal fires, the matched decision activates its `fast_response` plugin, which short-circuits the pipeline and returns an OpenAI-compatible 200 OK response (supporting both streaming SSE and non-streaming JSON) without forwarding to any upstream model.

7.5 Composable Safety Design

Placing safety detection in the signal layer enables a class of flexible security policies that are difficult to express in a serial plugin architecture. We highlight three patterns enabled by this design:

Context-aware thresholds. A single deployment can define multiple jailbreak rules at different sensitivity levels (e.g., $\theta = 0.4$ for strict, $\theta = 0.65$ for standard). Decisions compose these with domain signals: a `block_jb_finance` decision uses the strict rule AND the "economics" domain signal, while a `block_jb_general` decision uses the standard rule alone. This provides differentiated security posture without separate deployments.

Per-domain PII policies. Different domains require different PII handling: a healthcare endpoint may allow email and phone number for appointment booking while blocking SSN,
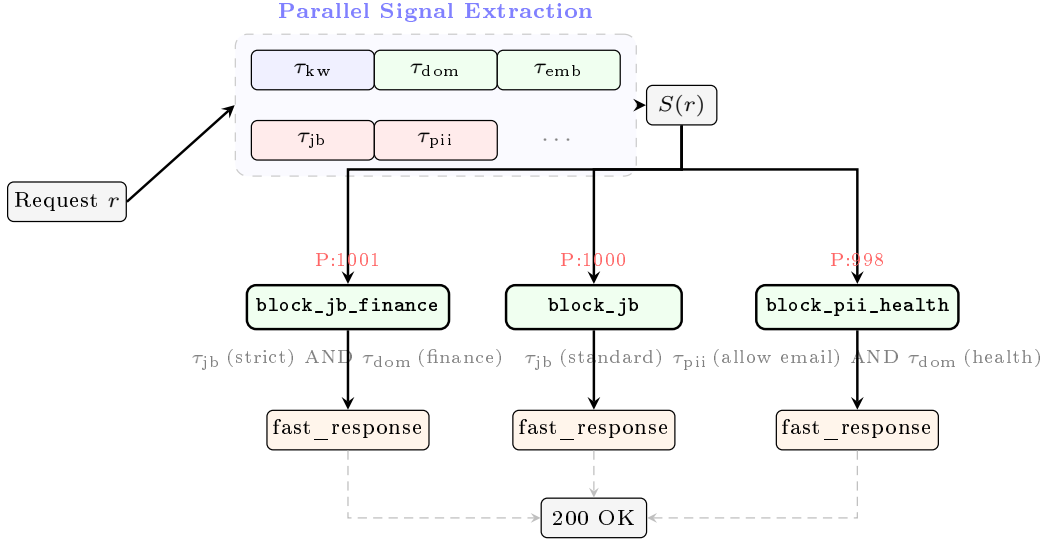
Figure 15: Signal-driven safety architecture. Safety classifiers (τ_{jb}, τ_{pii}) run in the signal layer in parallel with intent signals, adding zero marginal latency. Decision rules compose safety signals with domain signals using AND/OR logic, enabling context-aware policies (e.g., strict jailbreak detection only for finance, relaxed PII policy for healthcare). High-priority safety decisions activate the `fast_response` plugin to return an immediate 200 OK without contacting any upstream model.

whereas a public chatbot blocks all PII types. Each PII signal rule carries its own allow-list, and decisions combine the appropriate rule with the relevant domain signal. This replaces a monolithic PII policy with a set of composable, domain-scoped rules.

Graduated response. By defining multiple safety decisions at different priorities, the system can implement graduated responses: a high-confidence jailbreak ($c \geq 0.9$) triggers an immediate block (priority 1001), while a moderate-confidence detection ($0.4 \leq c < 0.9$) triggers a warning header without blocking (priority 500). The `fast_response` plugin handles blocking; a header-only decision allows the request to proceed with an observability annotation. This graduated model reduces false-positive impact while maintaining security coverage.

8   HALUGATE: GATED HALLUCINATION DETECTION

Hallucination—generating plausible but unsupported content—is a fundamental limitation of autoregressive language models Manakul et al. (2023); Min et al. (2023). We introduce HaluGate, a three-stage pipeline (Figure 16) 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.

8.1   DESIGN RATIONALE

Existing approaches apply hallucination detection uniformly to all responses Manakul et al. (2023) or require multiple response samples Min et al. (2023). 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.
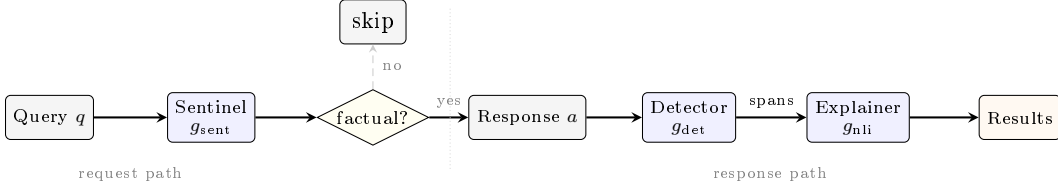
Figure 16: HaluGate three-stage gated pipeline. The Sentinel classifies queries on the request path; non-factual queries (40–60%) skip verification entirely (dashed). For factual queries, the Detector identifies hallucinated spans in the model response, and the Explainer provides NLI-based diagnostics per span.

8.2 Three-Stage Pipeline

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

$$g_{\text{sent}}(q) \in \{\text{NEEDS\_FACT\_CHECK}, \text{NO\_FACT\_CHECK}\} \tag{24}$$

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:

$$g_{\text{det}}(q, \mathbf{c}, a) = \left\{ (i, j, c_{ij}) \mid a_i \ldots a_j \text{ is unsupported by context } \mathbf{c} \right\} \tag{25}$$

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 Williams et al. (2018) classifies the relationship between the span and the grounding context:

$$g_{\text{nli}}(a_{i:j}, \mathbf{c}) \in \{\text{ENTAILMENT}, \text{CONTRADICTION}, \text{NEUTRAL}\} \tag{26}$$

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

8.3 Cost Analysis

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

$$\mathbb{E}[\text{Cost}] = C_{\text{sent}} + p_{\text{factual}} \cdot \left(C_{\text{det}} + \bar{k} \cdot C_{\text{nli}} \right) \tag{27}$$

Since the Sentinel is a lightweight LoRA-adapted classifier (Section 9) 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.

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.

8.4 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.

9 LoRA-BASED 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.

9.1 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:

$$M_{\text{indep}} = n \cdot |\theta_{\text{base}}| \tag{28}$$

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.

9.2 SOLUTION: SINGLE BASE MODEL WITH LoRA ADAPTERS

Low-Rank Adaptation (LoRA) Hu et al. (2022) represents task-specific weight modifications as low-rank decompositions:

$$W_i' = W + \Delta W_i = W + B_i A_i, \quad B_i \in \mathbb{R}^{d \times r}, \ A_i \in \mathbb{R}^{r \times d} \tag{29}$$

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:

$$M_{\text{LoRA}} = |\theta_{\text{base}}| + \sum_{i=1}^{n} 2rd = |\theta_{\text{base}}| + n \cdot 2rd \tag{30}$$

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.

31

Memory reduction.

$$\frac{M_{\mathrm{LoRA}}}{M_{\mathrm{indep}}} = \frac{|\theta_{\mathrm{base}}| + n \cdot 2rd}{n \cdot |\theta_{\mathrm{base}}|} \approx \frac{1}{n} \quad \text{for } 2nrd \ll |\theta_{\mathrm{base}}| \tag{31}$$

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

9.3 INFERENCE ARCHITECTURE

Each classification task proceeds as a full forward pass through the base model with the task-specific LoRA perturbation applied (Figure 17):

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.
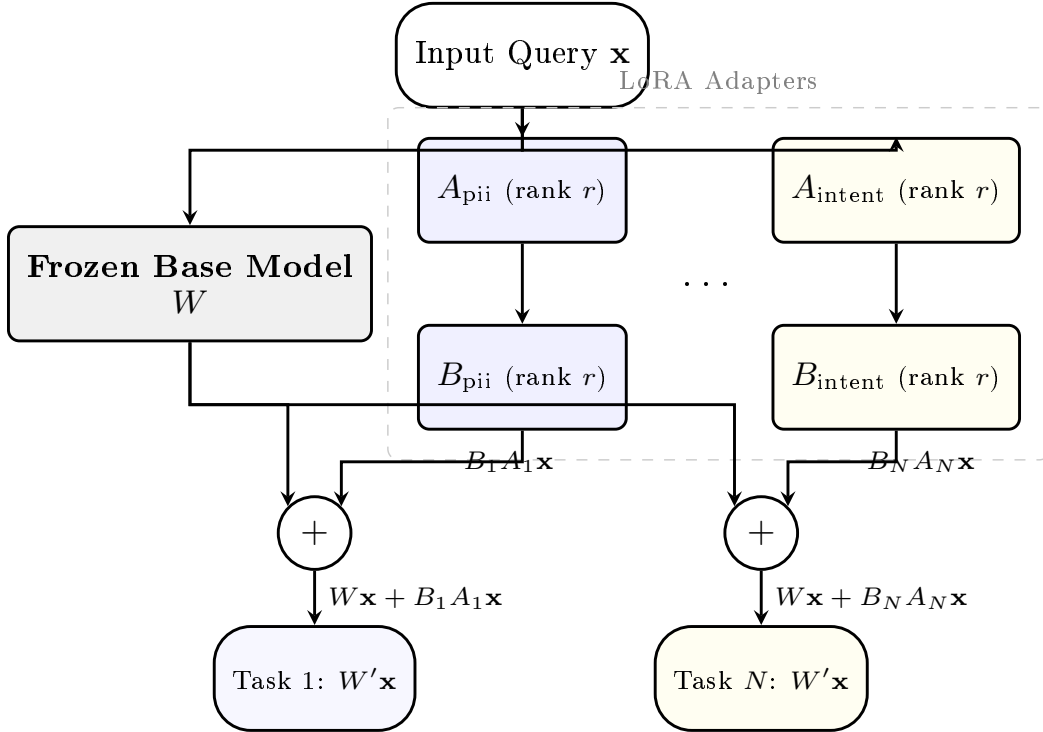

Figure 17: LoRA-based MoM inference architecture. The input query $\mathbf{x}$ is processed by a single frozen base model (W). Each task-specific LoRA adapter pair (A_i, B_i) computes a low-rank perturbation $B_i A_i \mathbf{x}$. The base output and adapter perturbation are summed to produce task-specific classifications, enabling multi-task support with minimal memory overhead.

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.

9.4 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 Warner et al. (2024) 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.

9.5 Training Methodology

All LoRA adapters are trained using PEFT Mangrulkar et al. (2022) 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.

10 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.
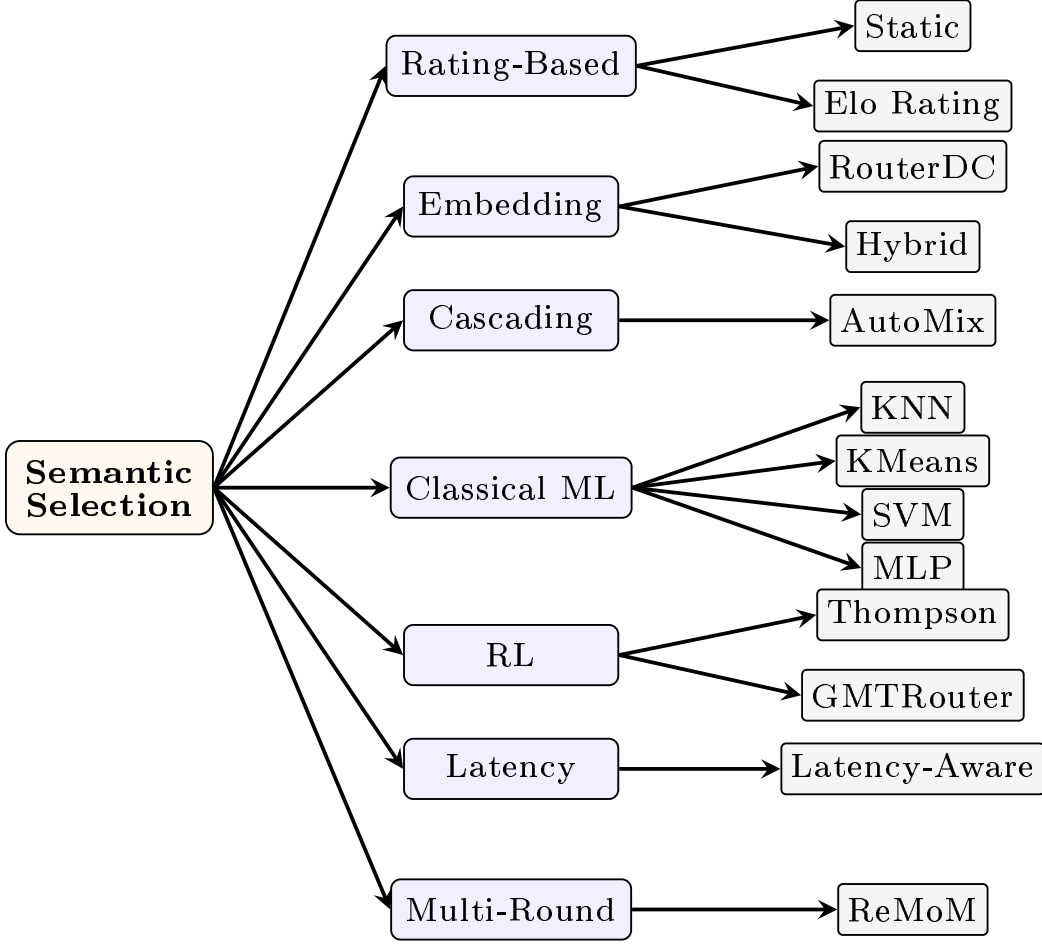
Figure 18: Taxonomy of thirteen semantic model selection algorithms organized by selection mechanism. Families span from lightweight rating-based methods (Static, Elo) to learned approaches (RouterDC, Classical ML, RL), adaptive cascading (AutoMix), real-time latency tracking, and multi-round synthesis (ReMoM).

10.1 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:

$$m^* = \arg \max_{m_k \in \mathcal{M}_{d^*}} \text{Quality}(\mathbf{e}_q, z, m_k; \Theta) - \lambda \cdot \text{Cost}(m_k) \tag{32}$$

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 families based on their selection mechanism (Figure 18).

10.2 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 Ong et al. (2024)). Models maintain Elo ratings R_k updated from pairwise user preference feedback. Selection probability follows the Bradley-

Terry model:

$$P(m_i \succ m_j) = \frac{1}{1 + 10^{(R_j - R_i)/400}} \tag{33}$$

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

10.3 Embedding-Based Selection

RouterDC Chen et al. (2024). Dual contrastive learning trains query and model encoders to produce embeddings in a shared space. Selection maximizes cosine similarity:

$$m^* = \arg\max_{m_k \in \mathcal{M}_{d^*}} \cos(\mathbf{e}_q, \mathbf{e}_{m_k}) \tag{34}$$

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

Hybrid Hu et al. (2024). Combines Elo ratings, embedding similarity, and cost in a weighted score:

$$\text{score}(m_k) = \alpha \cdot \tilde{R}_k + \beta \cdot \cos(\mathbf{e}_q, \mathbf{e}_{m_k}) + \gamma \cdot (1 - \tilde{c}_k) \tag{35}$$

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

10.4 Cascading Selection

AutoMix Aggarwal et al. (2023). 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:

$$\mathbb{E}[C] = \sum_{k=1}^{K} C_k \cdot \prod_{j=1}^{k-1} (1 - P(\hat{q}_j \geq \tau_j)) \tag{36}$$

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.

10.5 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:

$$\mathbf{f} = [\mathbf{e}_q \in \mathbb{R}^d; \ \text{onehot}(z) \in \{0, 1\}^C] \tag{37}$$

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

$$m^* = \arg\max_m \sum_{i \in \text{kNN}(\mathbf{f})} \mathbf{1}[m^{*(i)} = m] \cdot q^{(i)} \tag{38}$$

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

$$m^* = \arg\max_m \big(\alpha \cdot \text{quality}(m, z_{\text{cluster}}) - (1 - \alpha) \cdot \text{latency}(m)\big) \tag{39}$$

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:

$$P(m_k \mid \mathbf{f}) = \text{softmax}\big(W_2 \cdot \text{ReLU}(W_1\mathbf{f} + b_1) + b_2\big)_k \tag{40}$$

The MLP is implemented in the GPU-accelerated Candle runtime for low-latency inference.
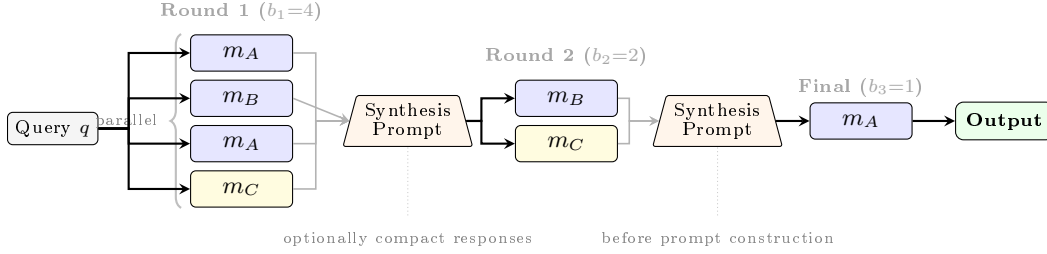
Figure 19: ReMoM execution flow with breadth schedule $\mathbf{b} = [4, 2]$. Round 1 distributes 4 parallel calls across models (m_A, m_B, m_C); responses are (optionally compacted and) assembled into a synthesis prompt. Round 2 sends 2 parallel synthesis calls. A final round ($b_3 = 1$, auto-appended) produces the single output. Each synthesis prompt includes the original query and all previous-round responses as numbered references, delegating quality judgment to the synthesizing LLM.

10.6 REINFORCEMENT LEARNING SELECTION

Thompson Sampling Thompson (1933). Each model maintains a Beta prior:

$$\theta_k \sim \text{Beta}(\alpha_k, \beta_k) \tag{41}$$

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 Xie et al. (2025). Models multi-turn user-query-model interactions as a heterogeneous graph. Graph neural network message passing captures complex interaction patterns:

$$\mathbf{h}_v^{(l+1)} = \text{AGG}\big(\{\mathbf{h}_u^{(l)} \mid u \in \mathcal{N}(v)\}\big) \tag{42}$$

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

10.7 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:

$$s_k = \frac{1}{|P|} \sum_{p \in P} \frac{\text{perc}_p(m_k)}{\min_j \ \text{perc}_p(m_j)} \tag{43}$$

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.

10.8 MULTI-ROUND REASONING (REMOM)

The ReMoM (Reasoning for Mixture of Models) strategy extends single-shot selection to multi-round parallel reasoning with LLM-driven synthesis. Inspired by PaCoRe Zhang et al. (2025b) but generalized to heterogeneous model pools, ReMoM executes a *breadth schedule* of decreasing parallelism across rounds, where each subsequent round synthesizes the outputs of the previous round via prompted LLM calls (Figure 19).

Breadth schedule. The operator specifies a breadth schedule $\mathbf{b} = [b_1, b_2, \ldots, b_R]$ defining the number of parallel model calls per round. A final synthesis round with $b_{R+1} = 1$ is automatically appended, yielding a total of $R+1$ rounds. For example, $\mathbf{b} = [32, 4]$ produces

three rounds: 32 parallel calls, then 4 parallel calls each synthesizing the 32 responses, then a single final call synthesizing the 4 responses.

Model distribution. At each round, b_r calls are distributed among the candidate models $\mathcal{M}_{d^*}$ according to one of three strategies: (1) *equal*: calls are distributed evenly across all candidates with round-robin remainder allocation; (2) *weighted*: calls are distributed proportionally to model weights (currently equivalent to equal distribution); (3) *first_only*: all b_r calls target a single model with different random seeds, providing PaCoRe-compatible single-model diversity. Calls within each round execute concurrently with configurable concurrency limits.

LLM-driven synthesis. After collecting responses from round r, the system constructs a synthesis prompt for round $r + 1$ using a configurable Go `text/template`. The default template presents the original query alongside all previous-round responses as numbered references, instructing the next-round model(s) to *"analyze these references and provide your own comprehensive solution."* When reasoning content is available (e.g., from models supporting extended thinking), the template additionally includes each reference's chain-of-thought reasoning. This approach delegates quality judgment entirely to the synthesizing LLM rather than relying on explicit scoring or weighted aggregation.

Response compaction. To manage prompt length across rounds, responses can be compacted before inclusion in synthesis prompts. Two strategies are supported: *full* (no compaction, the default) and *last_n_tokens* (retaining only the final N tokens, estimated at $\sim$4 characters per token). This is particularly important for high-breadth schedules where concatenating all responses would exceed context limits.

Execution flow. The complete algorithm proceeds as:

1. **Schedule construction**: Append [1] to the user-specified breadth schedule **b**.
2. **Round 1 (parallel generation)**: Distribute b_1 calls across candidate models; execute concurrently with temperature T (default 1.0) for response diversity.
3. **Rounds $2 \ldots R{+}1$ (synthesis)**: For each subsequent round, build a synthesis prompt from the previous round's (optionally compacted) responses, distribute b_r calls, and execute concurrently.
4. **Final output**: Return the single response from the final round ($b_{R+1} = 1$).

ReMoM is particularly effective when model capabilities are uncertain or when the task benefits from diverse perspectives (e.g., complex reasoning, multi-faceted analysis). The breadth schedule provides fine-grained control over the cost–quality tradeoff: higher initial breadth increases diversity at the cost of additional LLM calls, while the funneling structure ensures convergence to a single synthesized answer.

10.9   UNIFIED SELECTION INTERFACE

All thirteen algorithms implement a common interface:

$$\text{Select} : (\mathbf{e}_q, z, \mathcal{M}, \Theta) \to (m^*, c) \tag{44}$$

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.

10.10   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 12.3). 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.

11 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.

11.1 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 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).

11.2 Four-Runtime Architecture

We implement four inference runtimes, each optimized for different hardware and task profiles, all exposed to the Go routing process via C FFI and CGo (Figure 20):

Table 3: Inference runtime characteristics

Runtime	Target Hardware	Tasks	Framework
Candle	GPU (CUDA), CPU	Classification, LoRA, MLP	HF Candle Hugging Face (2024)
Linfa	CPU only	KNN, KMeans, SVM	Linfa Linfa Contributors (2024)
ONNX RT	CPU, GPU	Embeddings	ONNX Runtime Microsoft (2024)
NLP Binding	CPU only	BM25, N-gram matching	bm25 + ngrammatic (Rust)

All runtimes are compiled as Rust shared libraries (`.so`/`.a`) 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.

11.3 Candle Runtime: GPU-Accelerated Classification

The Candle runtime handles all transformer-based classification tasks, including LoRA adapter loading and inference (Section 9).

Supported architectures. BERT Devlin et al. (2019), ModernBERT Warner et al. (2024) (with Flash Attention and GeGLU), mmBERT-32K (YaRN RoPE for 32K context), DeBERTa v3 (NLI), and feed-forward MLPs (model selection).
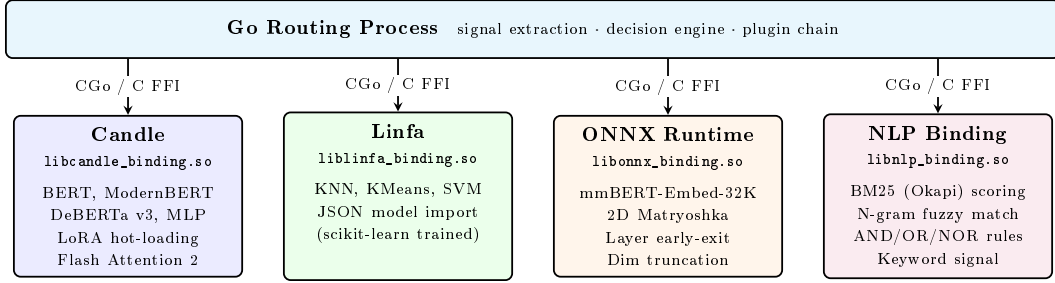
Figure 20: Four-runtime ML inference architecture. The Go routing process links to four Rust shared libraries via CGo/C FFI. Each runtime is specialized for a different class of ML workload, avoiding Python overhead on the critical path.

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.

11.4 MODERNBERT-BASE-32K: EXTENDED CONTEXT WINDOW

The Candle runtime supports ModernBERT-base-32k, extending the context window from 512 tokens (BERT-base) to 32,768 tokens via YaRN (Yet another RoPE extensioN) scaling Peng et al. (2023). This enables signal extraction from long documents and multi-turn conversations that exceed the traditional 512-token limit.

Architecture and scaling. ModernBERT-base-32k uses Rotary Position Embeddings (RoPE) with YaRN scaling to extend the base model's 8,192-token context to 32K tokens. The model maintains compatibility with existing LoRA adapters trained on BERT-base, enabling seamless migration without retraining classifier heads. Flash Attention 2 provides memory-efficient attention computation, reducing memory requirements from $O(n^2)$ to $O(n)$ for long sequences.

Performance characteristics. Empirical benchmarks on NVIDIA L4 GPU (23GB VRAM) demonstrate reliable performance for context lengths up to 8K tokens:

- **1K tokens**: p50 latency ∼94 ms at C=1, ∼970 ms at C=10 (100% success rate)
- **4K tokens**: p50 latency ∼955 ms at C=1, ∼9,389 ms at C=10 (93% success rate, 7 OOM errors)
- **8K tokens**: p50 latency ∼3,525 ms at C=1, fails at C=10 due to insufficient GPU memory

For sequences exceeding 32K tokens, automatic chunking with configurable overlap (default 128 tokens) enables processing of arbitrarily long documents while preserving context continuity.

Hardware requirements. Production deployments for 1K–8K tokens require a GPU with ≥23GB VRAM (e.g., NVIDIA L4, A10G). Full 32K token support and high concurrency (C=50+) require ≥40GB VRAM (e.g., NVIDIA A100). CPU inference is supported but incurs significant latency penalties (∼45× slower for 512 tokens).

Backward compatibility. The integration maintains full backward compatibility with existing BERT-base workflows. Sequences ≤512 tokens exhibit no performance degradation, and existing LoRA adapters (domain classification, PII detection, jailbreak detection) function without modification.

11.5 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.

11.6 ONNX Runtime: Efficient Embeddings

Embedding computation is served by ONNX Runtime, optimized for the mmBERT-Embed-32K model with 2D Matryoshka representation learning Kusupati et al. (2022).

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.

11.7 NLP Binding Runtime: Keyword Classification

The NLP Binding runtime handles BM25 and N-gram keyword matching for the keyword signal type, complementing the ML-based classifiers served by Candle. Unlike the neural runtimes, NLP Binding implements statistical text matching algorithms that require no model weights or GPU:

BM25 (Okapi). The BM25 classifier uses the Rust `bm25` crate to compute term-frequency–inverse-document-frequency scores between query text and keyword rules. Each rule specifies a set of keywords, a Boolean operator (AND/OR/NOR), a score threshold, and case-sensitivity. A keyword is considered matched when its BM25 relevance score exceeds the configured threshold.

N-gram fuzzy matching. The N-gram classifier uses the `ngrammatic` crate to perform fuzzy string matching via character n-gram overlap (default: trigrams, Jaccard similarity). This enables matching despite typos and morphological variation—e.g., `"urgnet"` matches the keyword `"urgent"` when the similarity exceeds the configured threshold.

FFI design. The binding follows the same conventions as `candle-binding`: Rust `#[repr(C)]` structs, `#[no_mangle] pub extern "C" fn` exports, CString-based string passing, and explicit free functions for Rust-allocated memory. The Go side wraps each classifier as a handle-based API (`BM25Classifier`, `NgramClassifier`) with lifecycle management (`New`, `AddRule`, `Classify`, `Free`). Thread safety is provided by Rust `Mutex`-guarded global state with atomic handle generation.

11.8 Runtime Selection Strategy

The routing system selects runtimes based on deployment configuration. The NLP Binding is always active when keyword signal rules use `bm25` or `ngram` methods:

- **GPU available**: Candle (classification + LoRA) + ONNX (embeddings) + Linfa (ML selection) + NLP Binding (keyword).
- **CPU only**: Candle with MKL (classification) + ONNX with early-exit (embeddings) + Linfa (ML selection) + NLP Binding (keyword).
- **Minimal**: ONNX (embeddings) + Linfa (ML selection) + NLP Binding (keyword), with classification delegated to external vLLM-served models.
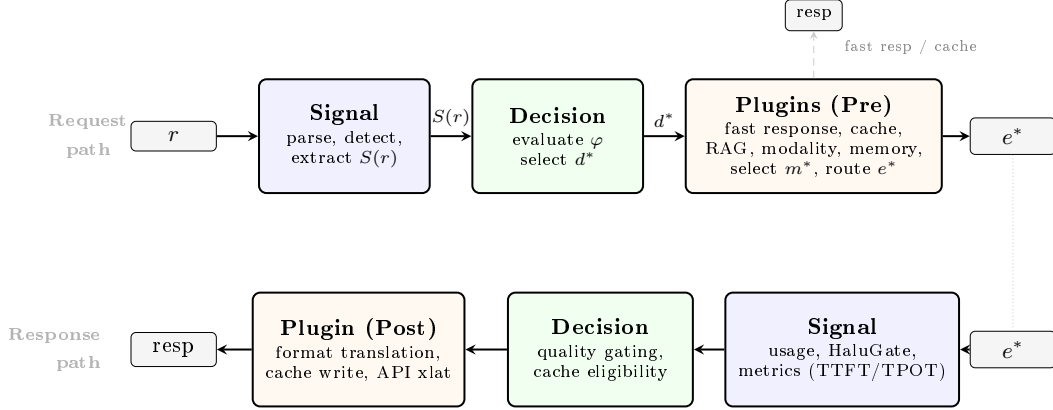
Figure 21: Bidirectional request processing pipeline aligned with the three-layer architecture. **Request path** (top, left→right): the signal layer parses and extracts $S(r)$, including safety signals (jailbreak, PII); the decision layer evaluates Boolean formulas to select d^*; the plugin chain executes pre-processing (fast response for safety enforcement, caching, RAG, memory), model selection, and endpoint routing. A cache hit or fast response short-circuits the pipeline (dashed). **Response path** (bottom, right→left): the same three layers operate in the same order—the signal layer extracts response-side signals (token usage, HaluGate hallucination scores, streaming latency metrics); the decision layer evaluates quality gating and cache eligibility; plugins perform format translation, cache writes, and API translation.

12 Request Processing Pipeline

We implement the routing system as an Envoy Envoy Proxy Authors (2024b) External Processor (ExtProc) Envoy Proxy Authors (2024a), enabling transparent interception of LLM API traffic without client-side modifications. Figure 21 illustrates the bidirectional request processing pipeline. This section describes the pipeline architecture, multi-provider routing, the Responses API integration, and the pluggable authorization factory.

12.1 Transparent Interception via ExtProc

The Envoy ExtProc protocol Envoy Proxy Authors (2024a) 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.

12.2 Request Body Pipeline

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

$$r \xrightarrow{\text{parse}} r' \xrightarrow{\text{signals}} S(r') \xrightarrow{\text{decide}} d^* \xrightarrow{\Pi_{\text{pre}}} \xrightarrow{\text{select}} m^* \xrightarrow{\text{route}} e^* \tag{45}$$

The stages execute in strict order: (1) API translation (Responses API → Chat Completions if applicable, see Section 12.4); (2) request parsing and provider detection; (3) signal extraction and decision evaluation (Sections 3 and 4); (4) fast response check (Section 5)— if the matched decision activates the `fast_response` plugin (e.g., for safety enforcement), the pipeline terminates with an immediate OpenAI-compatible response; (5) semantic cache lookup (Section 5)—cache hits terminate the pipeline with an immediate response; (6) RAG

context injection (Section 13); (7) modality routing (text vs. diffusion); (8) memory retrieval (Section 13); (9) model selection (Section 10), system prompt injection, and header mutation; (10) multi-endpoint resolution and provider-specific auth injection (Sections 12.3 and 12.5).

12.3 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 7 (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.

12.4 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 13), 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.

12.5 Authorization Factory

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

Definition 8 (Auth Provider). *An auth provider $\alpha : (Request, Endpoint) \to Headers'$ 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).

12.6 Response Body Pipeline

The response path performs: (1) token usage extraction for cost accounting; (2) format translation (provider-specific $\to$ OpenAI format); (3) streaming metrics computation (TTFT, TPOT); (4) hallucination detection via HaluGate (Section 8); (5) semantic cache writes for cache misses; (6) Responses API translation (Chat Completions $\to$ Responses API format, if applicable).

12.7 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.

13 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.

13.1 Persistent Memory

The memory system maintains user-scoped knowledge across conversation sessions, enabling personalized routing and context-aware responses. Figure 22 illustrates the full memory lifecycle.

Memory storage. Each conversation turn is stored directly as an *episodic chunk*—no external LLM is required. The user message and assistant response are concatenated into a `Q:/A:` block, sanitized (UTF-8 validation, 16 KB cap per entry), and embedded. A lightweight *entropy gate* discards turns that carry no retrievable signal—greetings, acknowledgments, and short one-word replies—before embedding, reducing index pollution. Every s turns a session-level sliding-window chunk spanning the last w turns is additionally stored (defaults: $s{=}3$, $w{=}5$), creating overlapping windows so facts near turn boundaries co-occur in at least one chunk—improving multi-hop retrieval coverage.

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 the hybrid search pipeline (vector similarity, BM25, and n-gram matching) over the user's memory store. An optional query-rewriting step reformulates the user's query for improved retrieval recall. Adaptive thresholding adjusts the similarity cutoff based on retrieval mode, preventing cosine-calibrated thresholds from silently discarding all RRF-scored results.

Post-retrieval filtering. Retrieved memories pass through a composable REFLECTION-GATE before context injection:

1. *Safety*: a regex block-list prevents prompt-injection payloads from being surfaced.
2. *Recency decay*: memories are weighted by recency, prioritizing recent context.
3. *Deduplication*: near-duplicate entries (high Jaccard similarity) are collapsed to a single representative.
4. *Budget*: the final set is capped at a configurable count to bound injected context length.

Filtered context is injected as a separate conversation message positioned after system instructions but before user turns, following the context-injection pattern of the OpenAI Agents SDK.

Background consolidation. A scheduled background job merges semantically related memories using greedy single-linkage clustering over word-level Jaccard similarity. Memories within each cluster are replaced by a single deduplicated summary entry, reducing store redundancy and improving retrieval precision over time.

13.2 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 11), and indexed in a vector store.

Hybrid retrieval. Pure vector search can miss lexically relevant results when the embedding model underweights rare terms. We implement a three-signal hybrid retrieval pipeline (Figure 23) that scores each candidate chunk along three axes:

1. *Vector similarity*: cosine similarity from the embedding index (already in $[0, 1]$).
2. *BM25*: an Okapi BM25 inverted index built over chunk contents provides keyword-level relevance, with configurable k_1 (term-frequency saturation, default 1.2) and b (length normalization, default 0.75) parameters.
3. *N-gram Jaccard*: character n-gram sets (default $n{=}3$) capture fuzzy lexical overlap, producing Jaccard similarity in $[0, 1]$.

Score fusion. Two fusion modes combine the three retriever scores into a single ranking: (1) *Weighted*: BM25 scores are min-max normalized to $[0, 1]$, then the final score is $w_v \cdot s_{\text{vec}} + w_b \cdot s_{\text{bm25}} + w_n \cdot s_{\text{ngram}}$ with configurable weights (defaults: 0.7, 0.2, 0.1). (2) *Reciprocal Rank Fusion (RRF)*: $\text{score}(d) = \sum_r 1/(k + \text{rank}_r(d))$, which is parameter-free beyond the constant k (default 60).
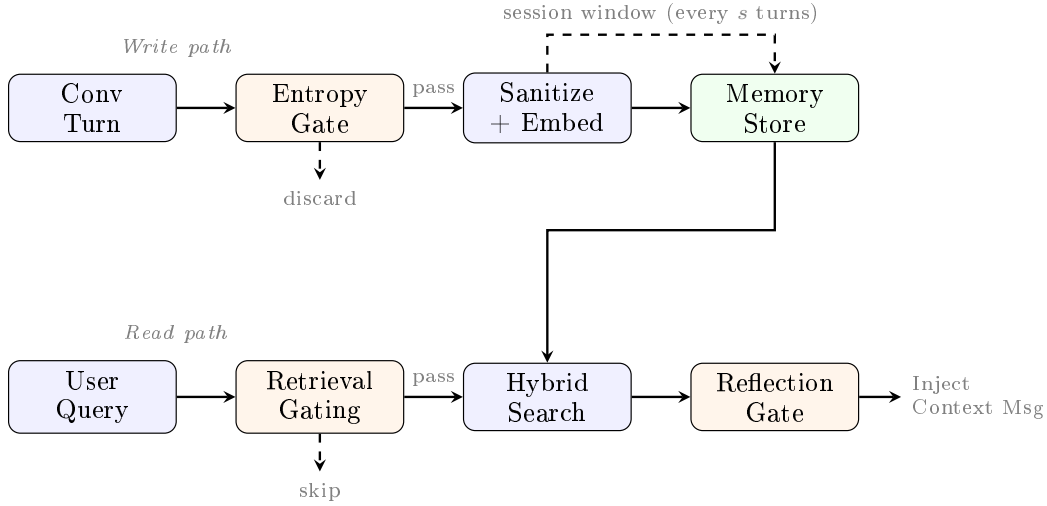
Figure 22: Memory lifecycle. *Write path*: each conversation turn passes an entropy gate (discarding low-signal turns such as greetings), is sanitized and embedded as an episodic `Q:/A:` chunk, and stored directly—no LLM inference required. Every s turns an additional sliding-window chunk is stored. *Read path*: a heuristic gate skips retrieval for irrelevant queries; qualifying queries run hybrid search (vector + BM25 + n-gram) over the user's store; retrieved memories pass through a REFLECTIONGATE (safety blocklist, recency decay, deduplication, budget cap) before injection as a separate conversation message.
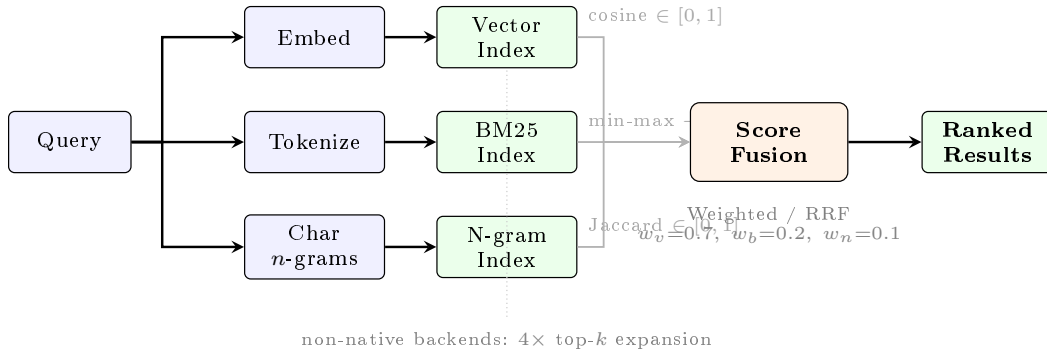

Figure 23: Hybrid search pipeline. Each candidate chunk is scored along three axes—vector cosine similarity, BM25 keyword relevance, and character n-gram Jaccard overlap—then combined via weighted fusion or Reciprocal Rank Fusion (RRF). Backends without native hybrid support fall back to a generic rerank path that fetches $4\times$ top-k candidates from vector search before applying BM25 and n-gram scoring.

Backend abstraction. The RAG plugin accesses vector stores through a common `VectorStoreBackend` interface, decoupling retrieval logic from storage implementation. Six backend types are supported (Figure 24):

- *In-memory*: development and testing; no external dependencies.
- *Milvus* Wang et al. (2021): production-grade distributed vector database with native hybrid search support.
- *Llama Stack*: delegates vector storage and search to a Llama Stack deployment via its OpenAI-compatible `/v1/vector_stores` API, enabling unified management of LLM serving and retrieval through a single platform. When the Llama Stack instance is configured with the Milvus `vector_io` provider, the backend supports hybrid search by passing

45

User Query → RAG Plugin → VectorStore Interface

Milvus — hybrid

Llama Stack — hybrid (via Milvus)

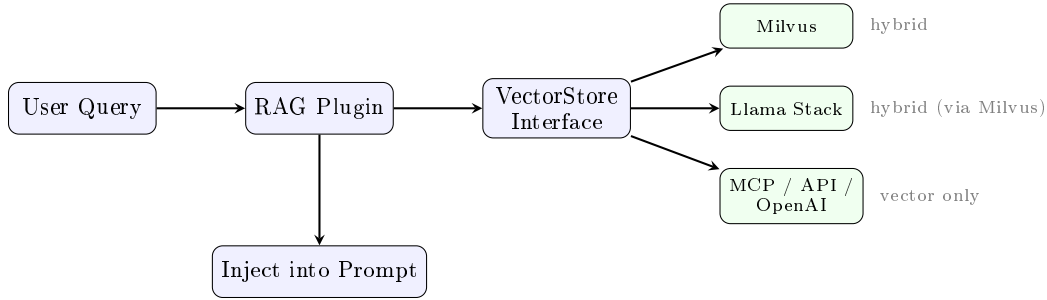MCP / API / OpenAI — vector only

RAG Plugin → Inject into Prompt

Figure 24: RAG backend architecture. The RAG plugin accesses vector stores through a common interface. Milvus and Llama Stack (with Milvus provider) support native hybrid search combining vector similarity with BM25 keyword matching via Reciprocal Rank Fusion. Other backends use vector-only retrieval with optional post-retrieval reranking.

`ranking_options: {ranker: "rrf"}` in search requests, combining vector similarity with BM25 keyword matching at the provider level.
- *External API*: any OpenAI-compatible vector store endpoint.
- *MCP*: retrieval via Model Context Protocol tool servers.
- *OpenAI file search*: delegated retrieval through OpenAI's hosted file search API.

Backends that implement native hybrid search (Milvus, Llama Stack with Milvus provider) use their own BM25 and keyword indexes; all other backends fall back to a generic rerank path that fetches an expanded candidate set ($4\times$ top-k) from vector search and applies BM25 and n-gram scoring as a post-retrieval reranking step.

Score-range awareness. Different retrieval modes produce scores on fundamentally different scales: cosine similarity yields values in $[0, 1]$ where a threshold of ~ 0.7 is typical, while RRF scores follow $\sum 1/(k + \text{rank})$ and typically range from 0.001 to 0.05. Applying a cosine-calibrated threshold to RRF scores would silently discard all results. The backend interface handles this transparently: when hybrid search is active, score-based filtering is bypassed and result volume is controlled solely by the top-k parameter.

13.3   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 without retransmitting the full message sequence. A bidirectional translator converts between the Response API format and Chat Completions format for backend model invocation, enabling all routing, safety, and caching features to operate identically on both API surfaces.

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 three backends: (1) *in-memory* for development; (2) *Redis* for production deployments requiring distributed state with high availability—supporting both standalone and cluster modes; (3) *Milvus* for deployments that benefit from semantic retrieval over conversation history. The Redis backend enables horizontal scaling: multiple router replicas share conversation state, and Redis persistence ensures that conversation chains survive pod restarts.

13.4   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, different search modes) or disable retrieval entirely. This enables, for example, a

"research assistant" decision that activates RAG with hybrid search over a technical knowledge base while a "casual chat" decision disables retrieval.

14 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.

14.1 METRICS TAXONOMY

We instrument four metric categories using Prometheus Prometheus Authors (2024):

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.

14.2 DISTRIBUTED TRACING

We implement OpenTelemetry OpenTelemetry Authors (2024) 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 Kwon et al. (2023) 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.

15 DEPLOYMENT

We describe the deployment architecture that enables the routing system to operate from single-node development to production Kubernetes The Kubernetes Authors (2024) clusters.

15.1 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.

15.2 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.

15.3 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.

16 Evaluation

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

16.1 Signal Extraction Latency

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

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

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 (~120 ms for domain classification) rather than the sum.

16.2 LoRA Memory Efficiency

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

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 ~6× less model memory (one base model + six ~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 9).

16.3 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.

16.4 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:

Table 6: Composable signal orchestration across deployment scenarios. Each scenario activates a different subset of the thirteen 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

16.5 End-to-End Routing Correctness

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

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

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.

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

16.7 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 25, 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.
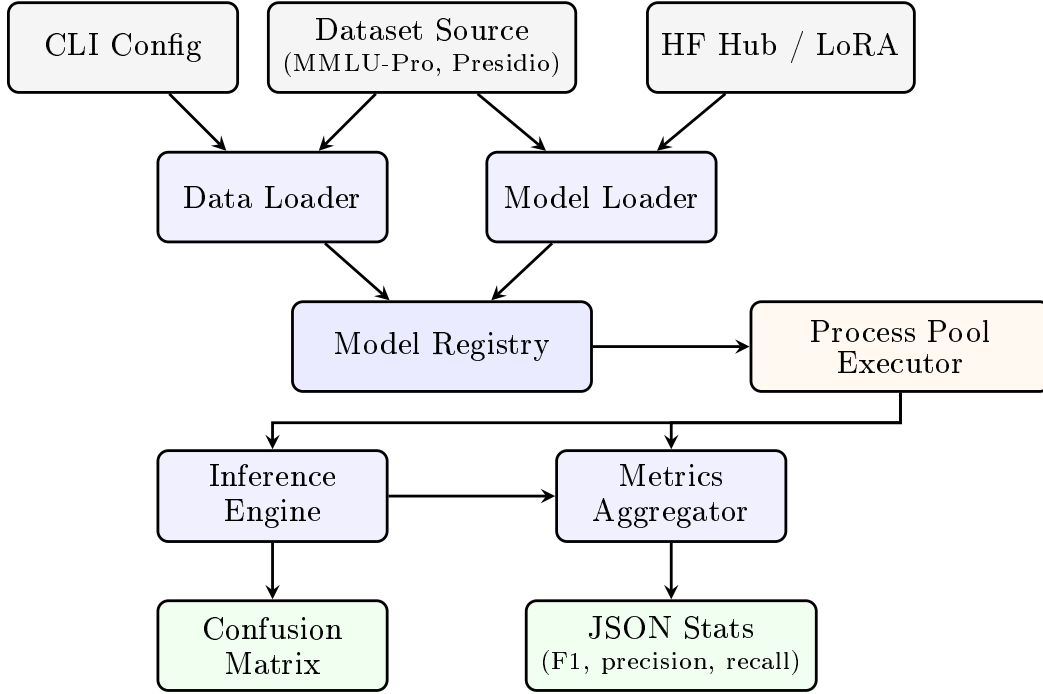
Figure 25: Unified MoM evaluation pipeline. Configuration and dataset sources feed into the data and model loaders, which populate the model registry. A process pool executor parallelizes inference across model variants, producing confusion matrices and aggregated metrics (weighted F1, per-class precision/recall) for both merged and LoRA models.

16.8 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.

17 RELATED WORK

17.1 LLM ROUTING AND MODEL SELECTION

Binary routing. RouteLLM Ong et al. (2024) 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 Chen et al. (2024) 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 Aggarwal et al. (2023) 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 Hu et al. (2024) proposed a benchmark for multi-LLM routing with hybrid scoring. Our Hybrid selector builds on this approach.

RL-based routing. Router-R1 Zhang et al. (2025a) formulates multi-LLM routing as a sequential decision process, using rule-based reward RL for multi-round routing and aggregation. GMTRouter Xie et al. (2025) 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.

17.2 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.

17.3 MIXTURE-OF-EXPERTS VS. MIXTURE-OF-MODELS

Sparse Mixture-of-Experts (MoE) Shazeer et al. (2017); Jiang et al. (2024) 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.

17.4 LLM SAFETY

Prompt injection defenses range from perplexity filtering and input preprocessing Jain et al. (2023) to fine-tuned safety classifiers Inan et al. (2023). PII detection systems Lison et al. (2021) 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.

17.5 HALLUCINATION DETECTION

SelfCheckGPT Manakul et al. (2023) detects hallucinations via multi-sample consistency. FActScore Min et al. (2023) 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.

17.6 SEMANTIC CACHING AND RAG

Semantic caching for LLMs uses embedding similarity to match incoming queries against previously seen requests, avoiding redundant model invocations. 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 Lewis et al. (2020); Wang et al. (2021) augments responses with retrieved context; our contribution is embedding RAG as a per-decision plugin within the routing framework.

18 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 evalua-

tion, 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.

18.1 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.

Differentiable entropy-folding policies. A natural extension of the layered entropy-folding view is to relax symbolic priority gates into trainable soft gates while preserving explicit policy constraints, enabling gradient-based optimization of control depth with auditable safety boundaries.

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.

Contrastive preference routing. The contrastive embedding method used for complexity classification generalizes to user preference learning: a contrastive preference classifier scores queries against exemplar sets representing different user preference profiles, enabling personalized model selection without per-user fine-tuning.

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. Preliminary work applies the contrastive embedding approach—already used for complexity classification—to multi-turn jailbreak detection: each user turn is scored against known-benign and known-adversarial exemplar sets, and accumulated turn-level signals are fused to detect gradual prompt escalation that evades single-turn classifiers.

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

Agent-based policy synthesis. The DSL configuration that specifies routing policies can be viewed as a program in a domain-specific language with a formally complete instruction set (Section 6.8). This reframes router configuration as a *program synthesis* problem: coding agents (LLMs fine-tuned for code generation) can translate natural-language routing specifications into valid DSL configurations, with routing quality feedback enabling RL-based optimization of the synthesis policy. Preliminary experiments suggest that this meta-learning approach—learning to *write* routing policies rather than to *make* individual routing decisions—can significantly reduce configuration effort while maintaining the interpretability and verifiability of symbolic decision rules.

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.

REFERENCES

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

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.

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

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.

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.

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.

Envoy Proxy Authors. Envoy external processing filter, 2024a. URL https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/ext_proc_filter.

Envoy Proxy Authors. Envoy proxy: An open source edge and service proxy, 2024b. URL https://www.envoyproxy.io/.

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

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.

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.

Hugging Face. Candle: Minimalist ML framework for Rust, 2024. URL https://github.com/huggingface/candle.

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

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.

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.

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.

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.

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.

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.

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*, pp. 661–670. ACM, 2010.

Linfa Contributors. Linfa: A Rust machine learning framework, 2024. URL https://github.com/rust-ml/linfa.

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)*, pp. 4188–4203, 2021.

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.

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. URL https://github.com/huggingface/peft.

Microsoft. ONNX Runtime: Cross-platform, high performance ML inferencing and training accelerator, 2024. URL https://onnxruntime.ai/.

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.

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.

OpenTelemetry Authors. OpenTelemetry: An observability framework for cloud-native software, 2024. URL https://opentelemetry.io/.

Bowen Peng, Eric Alcaide, Quentin Anthony, Amir Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Matteo Chung, Matteo Grella, et al. YaRN: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

Prometheus Authors. Prometheus: From metrics to insight, 2024. URL https://prometheus.io/.

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

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

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

Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723, 1938.

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

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.

The Kubernetes Authors. Kubernetes: Production-grade container orchestration, 2024. URL https://kubernetes.io/.

Alec Thomas. Participle: A parser library for Go. https://github.com/alecthomas/participle, 2024. Accessed: 2025-01-15.

William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

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.

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 finetuning and inference. *arXiv preprint arXiv:2412.13663*, 2024.

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

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

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

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

Tianjun Zhang, Yangjun Ruan, and Tatsunori Hashimoto. PaCoRe: Parallel cooperative reasoning with LLMs. *arXiv preprint arXiv:2601.05593*, 2025b.