XSR: RETHINKING WHERE SEMANTIC ROUTING HAPPENS

Anonymous authors
Paper under double-blind review

ABSTRACT

Semantic routing is an essential component of large-scale, multi large language model (LLM) serving, enabling requests to be dynamically directed to the right models based on their content. However, existing semantic routers operate as application-layer intermediaries, requiring requests to incur communication overhead before being forwarded to selected models. This design introduces severe overhead from application-layer processing and communication with external router processors. We present eXpress Semantic Router (XSR), a semantic router architecture that achieves high-throughput and low-latency by running latency-sensitive routing components, such as request processing, signal extraction, and decision policy, in the network layer. Our experiments show that across different routing functions (character n-gram, BM25, and distilled intent routing), XSR delivers up to 37.2× higher throughput across the evaluated concurrency ranges while reducing average latency by up to 79.7% compared to a state-of-the-art baseline, such as vLLM Semantic Router.

1 INTRODUCTION

As large language models (LLMs) vary widely in capability, latency, and cost, modern inference systems use semantic routing to dynamically select among heterogeneous models for each request (Liu et al., 2026). By selecting the optimal model for each request, semantic routing can balance response quality, latency, and inference cost (Ong et al., 2025). As the model ecosystem expands, routing becomes an increasingly important component of LLM inference.

However, enabling semantic routing for LLM inference adds additional processing overhead to the critical path of each request. For example, in vLLM Semantic Router (VSR), routing integrates Envoy through its External Processing (ExtProc) interface, which communicates with an external processor over a bidirectional gRPC stream (Liu et al., 2026; Envoy Project Authors, 2026). This design provides flexibility by allowing routing logic to inspect request content and make decisions outside Envoy, but it also adds userspace processing and communication before a request reaches its selected model. Because LLM inference systems must sustain high request rates while meeting strict latency constraints, semantic routing should add minimal overhead to the request path (Agrawal et al., 2024; Zhong et al., 2024).

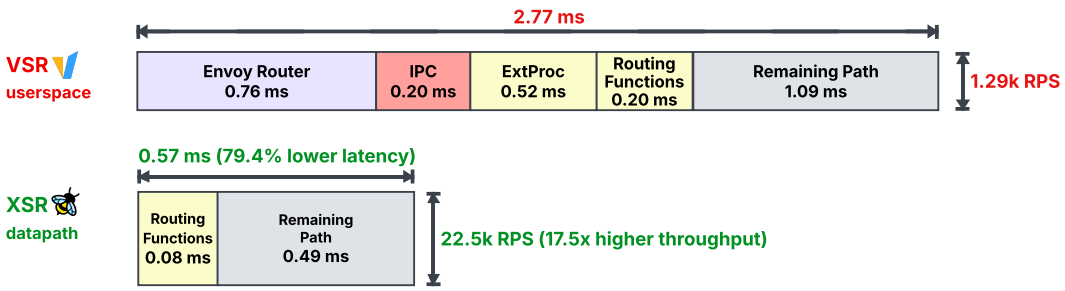


Figure 1: Mean latency is measured at 100 req/s, while throughput is measured at $c = 4$. Both paths use BM25 routing. XSR's router component is below 0.01 ms and is not shown separately. Section 4 details this decomposition.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

This overhead motivates a broader question: how can the overhead of semantic routing be reduced? A promising direction is to move routing functions into the network datapath, avoiding unnecessary application-layer processing and userspace communication. However, moving routing in the network datapath is not simply a matter of relocating an existing router, because routing functions vary in the data they consume, how they maintain state, and how they process each request (Shahinfar et al., 2023). Functions expressible as bounded keyword matching or Boolean rule evaluation are plausible candidates for direct offload, whereas learned classifiers and embedding-based similarity require substantially greater computation and model state (Liu et al., 2026). The resulting challenge is therefore how to reduce semantic routing overhead while preserving routing behavior.

Contributions. The main contribution of this work is named eXpress Semantic Router (XSR)[1], which is a network-level semantic routing architecture that treats routing as a function-placement problem, placing supported functions on the datapath to reduce userspace processing and communication overhead. Specifically, XSR's architecture consists of: (1) a function-placement approach that determines which routing functions can be directly offloaded, require a datapath-compatible realization, or remain in the application layer; (2) datapath-compatible implementations of character n-gram, BM25, and distilled intent routing with in-kernel policy evaluation and backend forwarding; and (3) an evaluation showing lower routing overhead than corresponding userspace paths and measuring routing agreement, with same-student implementation agreement evaluated separately from teacher-student agreement. In addition, Appendix A provides a summary of real-world impact on AI infrastructure.

1.1 Prior Work

Semantic routing for LLM inference. Most prior semantic routing systems focus on how routing decisions should be produced. Hybrid LLM routes queries according to predicted difficulty and target response quality (Ding et al., 2024), while RouteLLM learns routing policies from preference data to balance model quality and cost (Ong et al., 2025). AutoMix uses self-verification to determine when to escalate queries to larger models (Aggarwal et al., 2024), and MixLLM simultaneously considers response quality, cost, and latency when assigning queries to models (Wang et al., 2025). More recent systems such as BEST-Route and RADAR incorporate test-time compute or reasoning difficulty into routing decisions (Ding et al., 2025; Fernandez et al., 2026). VSR instead exposes multiple configurable request signals and decision rules for backend selection (Liu et al., 2026). These works mainly optimize model-selection policy, whereas XSR focuses on the architecture and execution placement of semantic routing.

Network kernel offload and eBPF feasibility. Extended Berkeley Packet Filter (eBPF) enables sandboxed programs to run within the Linux kernel, allowing selected application functionality to move into the kernel datapath. Shahinfar et al. (2023) argue that deciding what to offload requires considering not only whether code can satisfy eBPF's execution constraints but also its shared state, synchronization requirements, and expected performance benefit. Related work has addressed individual constraints of execution environments: K2 uses program synthesis to generate safe and efficient BPF programs under verifier restrictions (Xu et al., 2021), while Kumar et al. (2024) characterize the feasibility and limits of parsing application-layer messages in eBPF, and PinTok (Choi et al., 2026) is another work that shows great improvements by offloading tokenizer to the network kernel. XSR builds on these observations in the context of semantic routing, evaluating routing functions individually rather than transfering the entire userspace semantic router into the kernel.

Removing application-layer proxies from the critical path. Application-layer proxies provide flexible policy enforcement but can introduce substantial processing and inter-process communication overhead. MeshInsight demonstrates that service-mesh overhead can come from IPC and protocol parsing (Zhu et al., 2023), while ServiceRouter avoids such overhead by eliminating an additional proxy hop on the common RPC path (Saokar et al., 2023). Most closely related to XSR, Beeline moves supported L7 service mesh policies into an eBPF fast path, removing Envoy from the critical path while retaining the userspace proxy for unsupported policies (Brandner et al., 2026). XSR shares the motivation of avoiding unnecessary proxy processing, but targets semantic routing for LLM inference rather than general L7 service policies. In particular, XSR offloads prompt-

[1]The code will be open-sourced upon acceptance of the work.

derived routing functions to the network layer and evaluates both routing overhead and agreement with the corresponding userspace implementations.

2 METHOD

XSR treats semantic routing as four stages: request parsing, signal extraction, policy evaluation, and backend forwarding. We examine how functions in these stages can be realized in the network datapath. Functions that satisfy datapath constraints can be offloaded directly, while others may require a kernel-compatible reimplementation. Functions for which no such realization is available remain outside the current datapath.

The three evaluated routing functions illustrate different cases within design space: character-n-gram uses a datapath-compatible reimplementation, BM25 precomputes keyword-dependent scoring terms and performs request-time scoring, and learned intent routing uses a kernel-compatible approximation when the original model cannot execute in the datapath.

2.1 ARCHITECTURE

Overview of VSR request. In VSR, our baseline, Envoy invokes an external userspace semantic router through the External Processing (ExtProc) interface. The router extracts the relevant request content, computes routing signals, evaluates the configured policy, and returns routing metadata to Envoy, which applies it to select the upstream backend. Each routed request therefore requires a bidirectional gRPC exchange across the Envoy–Router boundary before forwarding. Section B provides a detailed view of request exchange between components, highlighting XSR's reduced number of request-path exchanges.

Router decomposition. General semantic routing, including the VSR baseline described above, can be explicitly decomposed into four request-path stages: request parsing, signal extraction, policy evaluation, and backend forwarding. Let R denote a complete plaintext HTTP request. A parser P extracts the inputs required by the configured signal functions, $Q = P(R) = \{q_1, \ldots, q_m\}$, where q_i denotes the derived input provided to signal function f_i. A configurable collection of signal functions $F = \{f_i\}_{i=1}^{m}$ evaluates these inputs according to each function's configured parameters. For each f_i, $s_i = f_i(q_i; \theta_i)$, where θ_i denotes its configured parameters. The resulting signal outputs form $S = F(Q) = \{s_1, \ldots, s_m\}$. A decision policy Π then selects a route $r = \Pi(S)$, and a route-to-backend mapping B resolves to a selected backend $b = B(r) \in \mathcal{B}$, where $\mathcal{B}$ denotes the set of configured backends. The decomposition can be summarized as

$$\underbrace{R}_{\text{request}} \xrightarrow{P} \underbrace{Q}_{\text{request inputs}} \xrightarrow{F} \underbrace{S}_{\text{routing signals}} \xrightarrow{\Pi} \underbrace{r}_{\text{selected route}} \xrightarrow{B} \underbrace{b}_{\text{backend}} .$$

Datapath Constraints. A routing function can be placed in the network datapath only when three requirements hold: (1) its required inputs are visible at the selected network hook, (2) its computation and state can be bounded to satisfy eBPF's execution and verification constraints, and (3) its datapath implementation does not require unavailable operations such as blocking I/O, userspace libraries, or external inference runtimes.

Together, these requirements define three function-placement cases: (a) functions that satisfy these constraints are directly offloaded to the network layer. (b) functions whose existing implementation does not satisfy these constraints, but whose behavior can be preserved with a datapath-compatible realization, are reimplemented for network execution. (c) functions that satisfy neither case remain in the application layer. The current XSR prototype implements the first two cases and does not provide a userspace fallback for unsupported functions.

Overall XSR routing architecture. XSR executes supported functions in the kernel socket datapath. After TLS termination, an `SK_SKB` stream parser receives the reassembled plaintext byte stream and waits for a complete HTTP request. It then extracts the request inputs Q required by the configured signal functions and evaluates F. The resulting signals S are evaluated by Π to select route r. XSR then resolves r to backend $b = B(r)$ and enforces the decision through SOCKMAP redirection, while the backend response is redirected to the client socket without proxy relay.

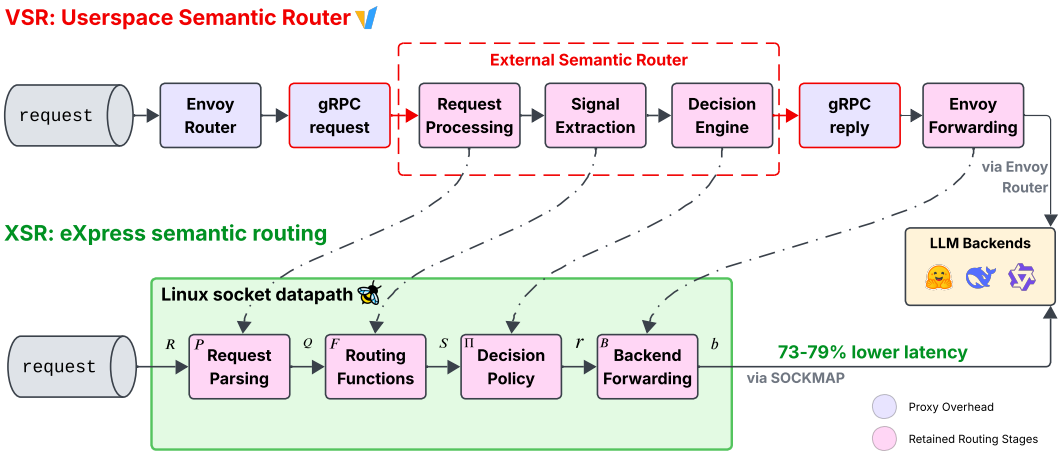


Figure 2: VSR and XSR routing architectures. XSR preserves the routing stages while eliminating ExtProc/gRPC exchange on each request.

2.2 Request Inputs and Routing Signals

Request input extraction. The first two stages of XSR transform a complete request R into the routing signals S,

$$R \xrightarrow{P} Q \xrightarrow{F} S.$$

Rather than directly providing individual packets to the routing functions, XSR operates on the reassembled plaintext byte stream and identifies complete HTTP requests before signal evaluation. The parser P extracts the request fields required by the configured signal functions, producing $Q = \{q_1, \ldots, q_m\}$. Each q_i contains the request input required by signal function f_i. For example, an OpenAI-compatible chat signal may operate on text from the JSON `content` field (OpenAI, 2026), while another signal may depend on an HTTP header. The socket datapath then provides these extracted inputs to their corresponding signal functions f_i for evaluation.

Routing signals. Each configured signal function f_i transforms its extracted request input q_i into a signal s_i according to configured parameter θ_i, producing a signal output

$$s_i = f_i(q_i; \theta_i).$$

The collection of outputs $S = \{s_1, \ldots, s_m\}$ contains the information made available to the routing policy for the current request. Depending on f_i, a signal s_i may indicate whether a configured condition matched or identify a semantic domain predicted by a learned classifier. More importantly, the function f_i that produces s_i is separate from the policy Π that subsequently determines the route. This separation allows XSR to implement each signal function according to the datapath, independent of route selection.

Keyword matching. Deterministic keyword-based signals provide a lightweight way to determine whether request content matches configured terms or phrases. Rather than relying on exact string matches, these functions can evaluate lexical similarity or relevance using methods such as character n-grams and BM25. Character n-gram similarity evaluates lexical overlap by decomposing the request input q_i and each configured keyword into overlapping sequences of n characters. XSR uses $n = 3$, following VSR's character-trigram configuration shown in Figure 3. BM25 provides a complementary realization based on term relevance rather than character-level overlap. For a configured keyword set K_i, VSR treats each keyword or phrase $k \in K_i$ as a document and uses the extracted request input q_i as the query. The resulting matched keywords can then be combined according to the configured keyword operator, such as AND, OR, or NOR, to produce the signal s_i. Further implementation details for character n-gram and BM25 matching are provided in Appendix C. VSR supports several such functions, but the experiments focus on character trigram similarity and BM25 as two representative realizations.

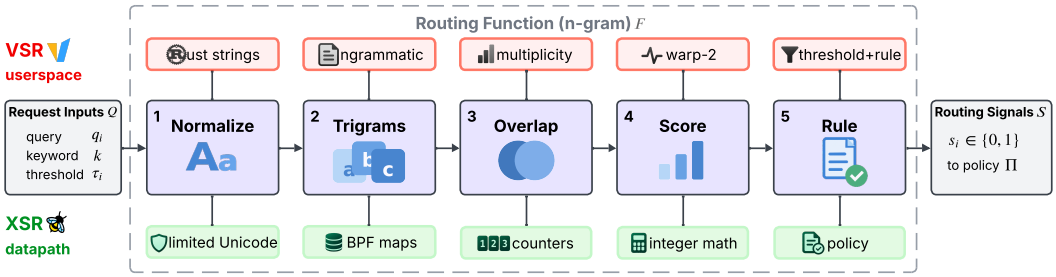


Figure 3: VSR and XSR implementations of character-trigram matching. XSR preserves the same matching stages while upholding eBPF constraints through integer arithmetic.

Model-based signals. Learned classifiers produce routing signals through model inference rather than deterministic lexical matching. In VSR, a 14-domain intent classifier based on mmBERT-32K maps request input q_i to a predicted intent label with an associated confidence score, which are exposed as a routing signal s_i for the decision policy.

Running the full transformer directly in the datapath is incompatible with XSR's execution design. XSR therefore uses mmBERT-32K as an offline teacher for a smaller 14-domain student whose representation and inference are designed for eBPF. At request-time, the student uses hashed byte trigrams and a linear classifier over a fixed-size feature space. Appendix D details the feasibility constraints, distillation procedure, and student implementation.

2.3 Route Selection and Backend Forwarding

Route selection. Routing signals describe request properties but do not determine the destination themselves. The decision policy Π evaluates the signal outputs S according to the configured routing rules and selects a route, $r = \Pi(S)$.

In VSR, the Decision Engine performs this stage by evaluating configured conditions over signal outputs. XSR preserves this separation between signal evaluation and route selection, but represents supported decision rules using bounded datapath structures. Boolean conditions over signals are encoded using fixed-width signal masks and rule entries, allowing combinations such as AND, OR, and NOT to be evaluated without recursive expression processing.

Backend Forwarding. Once route r is selected, the route-to-backend mapping B resolves it to a configured backend, $b = B(r)$ where $b \in \mathcal{B}$. The userspace control stores the route-to-backend mapping in BPF maps. The kernel then forwards the request to b using SOCKMAP redirection and redirects the corresponding backend response to the client socket.

2.4 Routing Correctness

Routing correctness evaluates how closely XSR preserves the routing behavior of the corresponding reference implementation when routing functions are moved or recreated in the datapath.

Equivalence criterion. Let $g_i(R; \theta) = r_i$ denote the route selected by system i for request R under configuration θ. θ includes the signal parameters, decision rules, route priorities, and backend mappings shared by the compared systems. Behavioral equivalence is defined as

$$g_{\text{XSR}}(R; \theta) = g_j(R; \theta) \qquad \forall R \in \mathcal{D},$$

The implementations may use different source code, data structures, and execution environments; for the same supported request and configuration, they must select the same route. The supported request domain and prototype bounds are detailed in Appendix F.

Learned-signal agreement. The signal produced by the distilled model differs from the exact implementations because the student approximates the mmBERT-32K rather than reproducing its computation exactly. For the distilled model-based signal, predicted domain between the student and the mmBERT-32K is compared under the shared decision policy.

Agreement metrics. We measure behavioral equivalence using request-level routing agreement. Each request is processed independently by XSR and the corresponding reference implementation under the same configuration, and the resulting routes are compared:

$$\text{Agreement}(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{R \in \mathcal{D}} \mathbf{1}\left[g_{\text{XSR}}(R;\theta) = g_j(R;\theta)\right].$$

Section 3 instantiates this architecture with representative lexical and learned routing signals and evaluates their routing performance and request latency.

3 EXPERIMENTS

We evaluate XSR using character n-gram, BM25, and distilled intent routing. We first measure agreement with the two userspace implementations, VSR and LLMRouter (LLMR) (Feng et al., 2026). We then compare request-path throughput and latency with Section 4 examining the sources of the performance differences. Deployment assumptions and experimental details are provided in Appendix E.

3.1 EXPERIMENTAL SETUP

Systems and testbed. All XSR, VSR, and LLMR receive the same request workloads and forward to the same lightweight backends. The backends mark and return fixed routes and identify responses, minimizing backend computation so the measurement primarily reflects the routing path. For intent routing, all three systems execute the same student classifier with 8,192 hashed features; VSR's default mmBERT-32K classifier is replaced only for this comparison.

Evaluated routing functions. We evaluate three routing functions: character n-gram, BM25, and a distilled 14-domain intent classifier. Character n-gram and BM25 represent two keyword-matching methods with different scoring behavior, while the intent classifier represents a model-based routing signal. For each function, we compare XSR with the corresponding userspace routing paths under the same request workloads and backend mappings. For the model-based signal, XSR and the userspace path execute the same distilled 14-domain student intent classifier.

Workloads. We use separate workloads for routing correctness and performance. For correctness, character n-gram and BM25 are evaluated using identical request set and routing configurations across the three routers, while the same labeled domain workload is used to evaluate the student and mmBERT-32K teacher. We report the agreement metrics defined in Section 3.2. For performance, we measure throughput as request concurrency increases and latency at a fixed request rate of 100 req/s. Each configuration is evaluated over five independent trials for both throughput and latency measurements; Appendix G provides the complete benchmark protocol and validity criteria.

3.2 ROUTING CORRECTNESS RESULTS

For keyword-matching functions, we measure exact routing agreement, while for the distilled intent classifier, we separately measure teacher–student agreement and implementation agreement. Complete agreement results are reported in Appendix H.1.

Same-model implementation. To fairly compare these systems, we evaluate XSR, VSR (Envoy+ExtProc), and LLMR using the same student classifier, replacing VSR's default mmBERT-32K classifier with the distilled student used by XSR. Under this controlled comparison, XSR selects the same route as both userspace implementations on all 1,400 requests. The XSR implementation also matches the userspace student reference on every prediction and all 14 integer class scores, showing that datapath execution introduces no additional disagreement for the evaluated model.

Model-based signal agreement. The distilled student approximates the mmBERT-32K teacher rather than reproducing its classification exactly. On the 1,400-request test set, the student matches the teacher's 14-domain prediction on 29.79% of requests. The deployed student therefore does

not preserve the teacher's classification quality on this workload. We use this result to demonstrate the feasibility of kernel-compatible learned inference rather than to claim parity with the teacher. Because 12 of the 14 domains map to the `others` route, route accuracy is dominated by the majority class. We therefore report route macro-F1 alongside accuracy in Table 1 to account for the uneven distribution of requests across routes.

Model	Domain acc.	Route acc.	Route macro-F1
mmBERT-32K teacher	62.14	91.86	64.39
Distilled 8K student	19.64	85.86	40.33
Always predict *others*	—	85.71	30.77

Table 1: Intent quality on the 1,400-request test set. All values are percentages vs. ground-truth labels. Route metrics use the shared three-route policy. The majority-route baseline is calculated from the test-set distribution; domain accuracy is not applicable as it predicts only a route.

The student's route accuracy is close to the majority-route baseline (85.86% vs. 85.71%), while its macro-F1 is higher (40.33 vs. 30.77), indicating improved minority-route discrimination despite limited overall classification quality.

Keyword-routing agreement. To evaluate keyword-routing correctness, we run 9,280 requests through XSR, VSR, and LLMR under the same configurations and backend mappings. The evaluation corpus consists of 880 SPEED-Bench requests and 8,400 RouterArena requests. Character n-gram routing matches across all three systems on all 9,280 requests. BM25 matches LLMR on all requests and VSR on 9,276 of 9,280 requests (99.96%). In the four BM25 mismatches, XSR agrees with an independent BM25 reference while VSR selects the default route. The mismatches persist across restarts and remain unresolved. These results apply only to the evaluated policies.

3.3 Routing Performance Results

We compare XSR's complete request path with userspace routing paths using throughput and latency metrics. The XSR–VSR comparison measures the difference between the XSR and Envoy+ExtProc request paths. For the intent signal, VSR and LLMR execute the same distilled 8K student as XSR, controlling for classifier choice. This control does not isolate kernel execution from differences in proxying, forwarding, and network topology.

Throughput across concurrency levels. Figure 4 shows XSR sustaining higher throughput than the userspace routers across all three routing functions. Relative to VSR, XSR achieves 6.76–12.98× higher throughput for character n-gram and 10.60–17.52× for BM25. For intent, XSR achieves 9.51–37.24× higher throughput than the same-model VSR (Envoy+ExtProc) path over the valid concurrency range of 1–16.

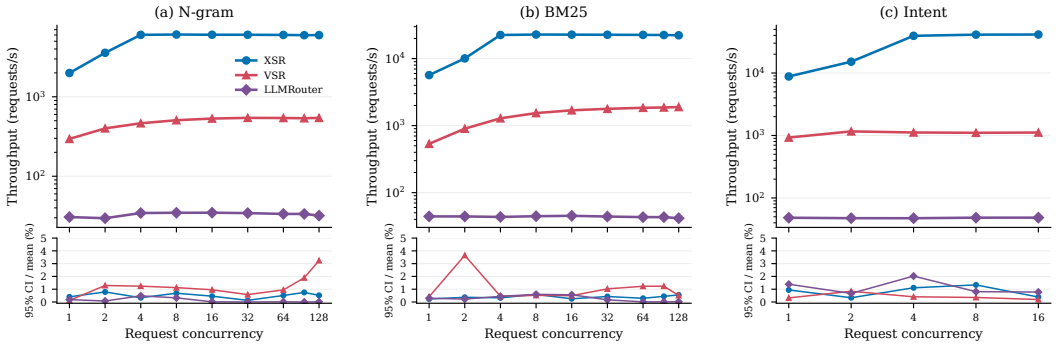


Figure 4: Evaluated throughput across concurrency levels for the three routing functions over five trials. The lower panels report 95% confidence intervals (CI) relative to the mean. Higher is better.

Fixed-rate latency. Figure 5 reports average and P99 latency at an offered request rate of 100 req/s. The steady-state comparison is XSR versus VSR; LLMR did not sustain this rate and is shown only for comparison. XSR reduces mean latency by 74.1% for character n-gram, 79.7% for BM25, and 73.1% for intent relative to the corresponding VSR/Envoy userspace path. P99 latency is reduced by 65.8%, 72.9%, and 71.3%, respectively. Complete results appear in Appendix H.

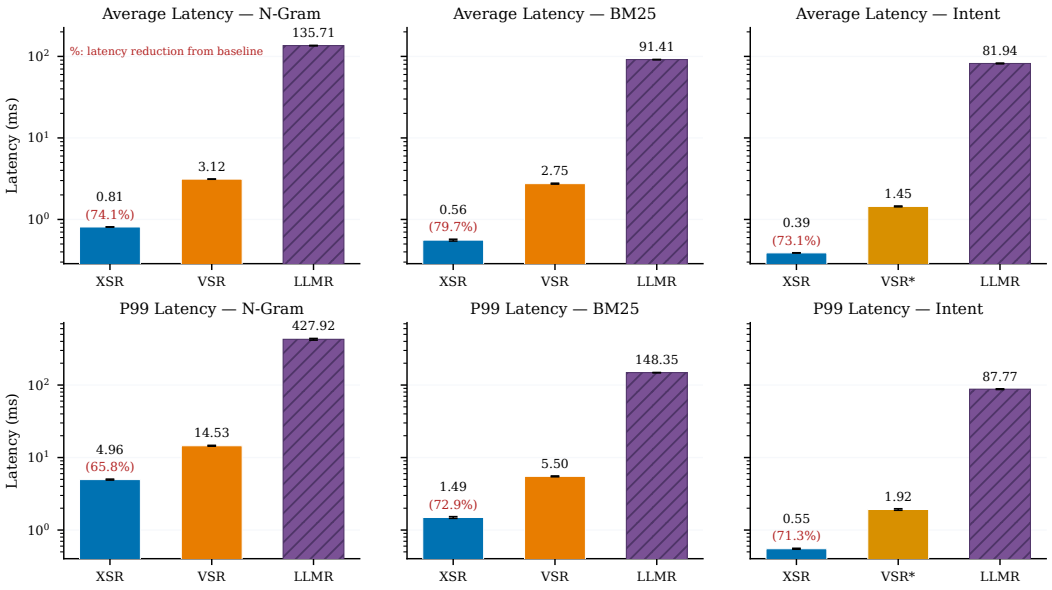


Figure 5: Average and P99 latency at 100 req/s over five trials with 95% CI. LLMR did not sustain the request rate, so striped bars show overloaded runs and are not valid latency results. VSR* uses the same student model as XSR through a separate userspace inference component. Lower is better.

4 Ablation Study

We use forwarding controls, component profiling, and a same-model comparison to examine the request-path costs underlying XSR's performance gain.

Effect of request-path components. We compare the BM25 routing paths with their forwarding controls at 100 req/s: direct forwarding for XSR and Envoy-only forwarding for VSR. Adding XSR's routing path increases mean latency by 0.164 ms over direct forwarding, whereas VSR's semantic-routing path increases it by 1.961 ms over Envoy-only. At P99, the corresponding increases are 0.850 ms and 4.295 ms. The Envoy–ExtProc routing path adds substantially more latency beyond its forwarding control than XSR's datapath routing path.

Request-path decomposition. We profile the BM25 paths at 100 req/s, as summarized in Figure 1. VSR's 2.77 ms total includes 0.76 ms of Envoy processing, 0.20 ms of Envoy–ExtProc communication, 0.52 ms of external-router processing, and 0.20 ms of BM25 evaluation; the remaining path accounts for 1.09 ms. XSR removes the Envoy–ExtProc exchange and executes routing in the datapath. Within its 0.57 ms total, BM25 evaluation together with request parsing accounts for 0.08 ms, while directly measured policy evaluation, backend selection, and SOCKMAP processing contribute less than 0.01 ms. These sub-0.01 ms costs are folded into the 0.49 ms remaining path shown in Figure 1. XSR also precomputes BM25 weights, reducing request-time scoring to integer accumulation and threshold comparison (Appendix C). The routing-function timings cover different scopes because XSR's measurement includes request parsing; they are not an isolated comparison of BM25 evaluation. Taken together, this decomposition shows that XSR's advantage reflects a restructured request path, including moving BM25 work out of request processing and eliminating the Envoy–ExtProc exchange, rather than an isolated speedup in BM25 evaluation.

Same-model comparison. The intent results in Figures 4 and 5 hold the 8K student classifier and routing policy fixed across XSR and the Envoy+ExtProc userspace path. XSR achieves 37.0× higher throughput at $c = 16$, and reduces mean and P99 latency by 73.1% and 71.3%, respectively, at 100 req/s. The advantage therefore persists without changing classifier choice, although the comparison measures complete request paths rather than execution placement alone.

5 Discussion and Limitations

Implications and future work. The results indicate that semantic-routing functions can avoid application-layer transitions when their computation can be represented within the datapath. More broadly, XSR suggests that reducing routing overhead can make semantic routing practical in increasingly latency-sensitive online systems, where latency added at each routing step can accumulate across repeated routing decisions. Future work includes a hybrid kernel datapath with a userspace slow path, trusted client- or upstream-generated routing hints, routing caching, support for additional routing signals, dynamic backend-health and load information, and evaluation with production model servers across separate hosts. In addition, another line of future work includes utilizing dedicated network hardware to perform semantic routing, entirely bypassing the host.

Limitations. XSR currently supports only routing functions and request inputs that can be implemented within datapath constraints. The supported input domain and prototype constraints are detailed in Appendix F. For learned routing, the deployed student sacrifices classification quality relative to the mmBERT-32K teacher, as quantified in Section 3.2 and Appendix D.

Header-assisted routing. Header-based routing is a well-established Layer-7 mechanism: HTTP load balancers can make backend-selection decisions from request headers and other application-level metadata (Elson & Howell, 2008), while service-mesh architectures support custom routing based on attributes such as host headers, URL paths, and API versions, with routing policy configured by the control plane and enacted by data-plane proxies (El Malki & Zdun, 2019). Existing AI gateways expose similar mechanisms for explicitly selecting models or providers from request metadata (WSO2, 2026; Kong Inc., 2026). When a trusted upstream component can compute the routing decision, it could therefore encode the selected route as a routing hint, leaving the gateway to perform only metadata-based selection rather than semantic classification. Systems such as HardMesh (You et al., 2025) have also explored moving ingress processing and routing into programmable network hardware, suggesting that XSR could support the same pattern directly in the datapath.

6 Conclusion

In this work, we present XSR, a network-level semantic routing architecture that accelerates routing by moving supported request-path functions from an external userspace router into the kernel socket datapath. Our prototype implements character-n-gram and BM25 keyword-matching functions, a distilled intent classifier, Boolean policy evaluation, and backend forwarding within eBPF. Across character n-gram, BM25, and distilled intent routing, XSR delivers 6.8–37.2× higher throughput across the evaluated concurrency ranges while reducing average latency by 73.1–79.7% at 100 req/s compared with VSR's Envoy+ExtProc userspace request path.

Beyond the current prototype, XSR suggests that semantic routing does not need to be treated as a single application-layer service. Routing functions that satisfy datapath constraints can be moved into the network path for immediate overhead reductions, while more expensive functions may continue to rely on userspace execution. As LLM serving models reduce latency and increase throughput, routing overhead can become an increasingly important part of the request path. Our next step is to extend XSR beyond the current prototype by studying how a broader range of routing functionality can benefit from datapath execution, including additional routing functions, protocol support beyond HTTP/1.1, and hybrid kernel–userspace designs.

AI USE STATEMENT

We used AI tools to proofread and format the paper for readability. The research methodology and experimental design were developed by the authors. All AI-assisted text was reviewed by the authors, and all reported experiments and measurements were executed by the authors in the experimental environment described in this paper. The authors take responsibility for the final content and results of this work.

ETHICS STATEMENT

This work does not involve human-subject studies or live user traffic. Experiments use existing benchmark datasets and a separate evaluation dataset. Since XSR processes request content after TLS termination, deployments should restrict plaintext request handling to trusted infrastructure with appropriate access controls.

REPRODUCIBILITY STATEMENT

The XSR implementation and experimental code are provided as supplementary materials, along with a `README.md` file containing installation and benchmarking instructions. Appendix E describes the experimental environment, and Appendix G provides the benchmark protocol and reporting procedure used to reproduce the evaluation.

REFERENCES

Talor Abramovich, Maor Ashkenazi, Izzy Putterman, Benjamin Chislett, Tiyasa Mitra, Bita Darvish Rouhani, Ran Zilberstein, and Yonatan Geifman. SPEED-Bench: A unified and diverse benchmark for speculative decoding. arXiv preprint arXiv:2604.09557, 2026. URL https://arxiv.org/abs/2604.09557.

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. In *Advances in Neural Information Processing Systems*, volume 37, pp. 131000–131034, 2024. doi: 10.52202/079017-4164. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/ecda225cb187b40ea8edc1f46b03ffda-Abstract-Conference.html.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/agrawal.

Laurin Brandner, Ayush Mishra, Sebastiano Miano, Aurojit Panda, Gianni Antichi, and Laurent Vanbever. Enforcing application-layer policies in eBPF. arXiv preprint arXiv:2605.31084, 2026. URL https://arxiv.org/abs/2605.31084.

Gohar Irfan Chaudhry, Esha Choukse, Haoran Qiu, Inigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. Murakkab: Resource-Efficient agentic workflow orchestration in cloud platforms. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 567–587, Seattle, WA, July 2026. USENIX Association. ISBN 978-1-939133-55-7. URL https://www.usenix.org/conference/osdi26/presentation/chaudhry.

Sean Choi, Myungheon Chin, and Ernest K. Ryu. Pintok: Tokenizers deserve dedicated pinned CPU-compute and memory. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=n4tTOeRa0a.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/b47d93c99fa22ac0b377578af0a1f63a-Abstract-Conference.html.

Dujian Ding, Ankur Mallick, Shaokun Zhang, Chi Wang, Daniel Madrigal, Mirian Del Carmen Hipolito Garcia, Menglin Xia, Laks V. S. Lakshmanan, Qingyun Wu, and Victor Rühle. BEST-Route: Adaptive LLM routing with test-time optimal compute. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 13870–13884. PMLR, 2025. URL https://proceedings.mlr.press/v267/ding25d.html.

Amine El Malki and Uwe Zdun. Guiding architectural decision making on service mesh based microservice architectures. In *Proceedings of the 13th European Conference on Software Architecture (ECSA)*, volume 11681 of *Lecture Notes in Computer Science*, pp. 3–19. Springer, 2019. doi: 10.1007/978-3-030-29983-5_1. URL https://doi.org/10.1007/978-3-030-29983-5_1.

Jeremy Elson and Jon Howell. Handling flash crowds from your garage. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, pp. 171–184, Boston, MA, 2008. USENIX Association. URL https://www.usenix.org/conference/2008-usenix-annual-technical-conference/handling-flash-crowds-your-garage.

Envoy Project Authors. External processing, 2026. URL https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/ext_proc_filter. Envoy documentation snapshot, accessed August 15, 2026.

Tao Feng, Fangxu Yu, Haozhen Zhang, Zhongjie Dai, Liangqi Yuan, Zijie Lei, Weizhi Zhang, Kunlun Zhu, Haodong Yue, Keyang Xuan, Ge Liu, and Jiaxuan You. LLMRouter: Unified infrastructure for developing, evaluating, and deploying LLM routers. arXiv preprint arXiv:2608.06867, 2026. URL https://arxiv.org/abs/2608.06867.

Nigel Steven Fernandez, Branislav Kveton, Ryan Rossi, Andrew Lan, and Jack Wang. RADAR: Reasoning-ability and difficulty-aware routing for reasoning LLMs. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/b1f7288854d3bd476c17725c2d85967f-Abstract-Conference.html.

Ron Haberman, Huamin Chen, Ricardo Noriega De Soto, Andre Fredette, David Brewster, Shane Utt, and Christopher Ferreira. LLM semantic router: Intelligent request routing for large language models, May 2025. URL https://developers.redhat.com/articles/2025/05/20/llm-semantic-router-intelligent-request-routing. Red Hat Developer, published May 20, 2025. Accessed August 27, 2026.

Takanori Hara and Masahiro Sasabe. Practicality of in-kernel/user-space packet processing empowered by lightweight neural network and decision tree. *Computer Networks*, 240:110188, 2024. doi: 10.1016/j.comnet.2024.110188. URL https://doi.org/10.1016/j.comnet.2024.110188.

Kong Inc. AI Models, 2026. URL https://developer.konghq.com/ai-gateway/entities/ai-model/. Kong documentation, accessed September 3, 2026.

Ashwin Kumar, Abhik Bose, Khushboo Tiwari, Arnav Mishra, Abhishek Dixit, Abuhujair Khan, and Mythili Vutukuru. Feasibility of application layer header parsing in eBPF and P4. In *2024 IFIP Networking Conference (IFIP Networking)*, pp. 475–481, 2024. doi: 10.23919/IFIPNetworking62109.2024.10619855. URL https://doi.org/10.23919/IFIPNetworking62109.2024.10619855.

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, Xue (Steve) Liu, and Bowei He. vLLM semantic router: Signal driven decision routing for mixture-of-modality models. arXiv preprint arXiv:2603.04444, 2026. URL https://arxiv.org/abs/2603.04444.

Yifan (Louie) Lu, Rixin Liu, Jiayi Yuan, Xingqi Cui, Shenrun Zhang, Hongyi Liu, and Jiarong Xing. RouterArena: An open platform for comprehensive comparison of LLM routers. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/4987bb24bc53c198785922d1bd9e18cf-Abstract-Conference.html.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Agentix: An efficient serving engine for LLM agents as general programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 2443–2459, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/luo.

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 from preference data. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/5503a7c69d48a2f86fc00b3dc09de686-Abstract-Conference.html.

OpenAI. Chat completions API reference, 2026. URL https://developers.openai.com/api/reference/resources/chat. Accessed: 2026-09-03.

Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 969–985, Boston, MA, 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/saokar.

Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. Automatic kernel offload using BPF. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pp. 143–149. ACM, 2023. doi: 10.1145/3593856.3595888. URL https://doi.org/10.1145/3593856.3595888.

The Linux Kernel Documentation. BPF Design Q&A, 2026. URL https://docs.kernel.org/bpf/bpf_design_QA.html. Linux kernel documentation snapshot, accessed September 3, 2026.

Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys*, 53(1):16:1–16:36, 2020. doi: 10.1145/3371038. URL https://doi.org/10.1145/3371038.

vLLM Semantic Router. mmBERT-32K Intent Classifier (Merged Model), 2026. URL https://huggingface.co/llm-semantic-router/mmbert32k-intent-classifier-merged. Hugging Face model repository, accessed 2026-09-03.

vLLM Semantic Router Team. Signal-decision driven architecture: Reshaping semantic routing at scale, November 2025. URL https://vllm.ai/blog/2025-11-19-signal-decision. vLLM Blog, published November 19, 2025. Accessed August 27, 2026.

vLLM Semantic Router Team. mmBERT-32K-YaRN: Extended context modern multilingual BERT. Hugging Face model repository, 2026a. URL https://huggingface.co/llm-semantic-router/mmbert-32k-yarn.

vLLM Semantic Router Team. Router API reference, 2026b. URL https://vllm-sr.ai/docs/v0.2/api/router/. vLLM Semantic Router documentation, version 0.2. Accessed August 27, 2026.

Xinyuan Wang, Yanchi Liu, Wei Cheng, Xujiang Zhao, Zhengzhang Chen, Wenchao Yu, Yanjie Fu, and Haifeng Chen. MixLLM: Dynamic routing in mixed large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 10912–10922. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.545. URL https://aclanthology.org/2025.naacl-long.545/.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. In *Advances in Neural Information Processing Systems*, volume 37, pp. 95266–95290, 2024. doi: 10.52202/079017-3018. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/ad236edc564f3e3156e1b2feafb99a24-Abstract.html.

WSO2. LLM Header Routing, 2026. URL https://wso2.com/api-platform/docs/ai-gateway/next/routing/routing-policies/llm-header-routing/. WSO2 AI Gateway documentation, accessed September 3, 2026.

Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the ACM SIGCOMM 2021 Conference*, pp. 50–64, 2021. doi: 10.1145/3452296.3472929. URL https://doi.org/10.1145/3452296.3472929.

Myoungsung You, Jaehyun Nam, Minjae Seo, Taejune Park, and Seungwon Shin. HardMesh: Enabling high-performance service mesh ingress processing with SmartNICs. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pp. 1235–1237. ACM, 2025. doi: 10.1145/3718958.3750471. URL https://doi.org/10.1145/3718958.3750471.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pp. 142–157, 2023. doi: 10.1145/3620678.3624652. URL https://doi.org/10.1145/3620678.3624652.

A XSR's Potential Impact on Infrastructure

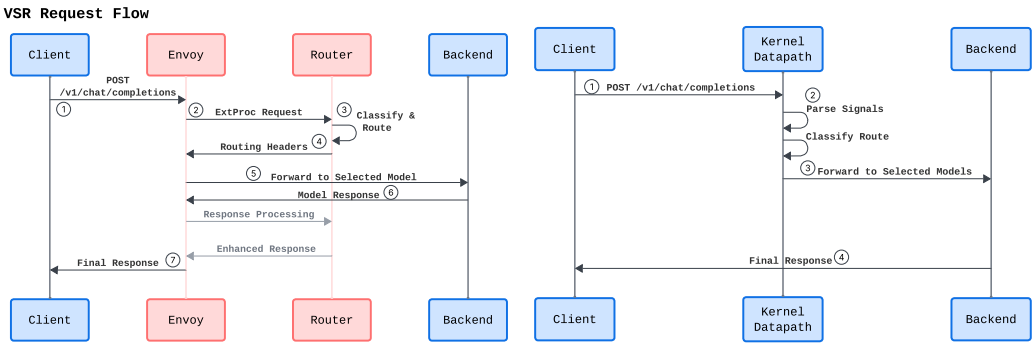
XSR's gains can translate into lower infrastructure cost for semantic routing at scale. Routing and proxy layers can themselves require substantial compute: MeshInsight reports service-mesh configurations with up to 163% more virtual CPU usage and 269% higher latency (Zhu et al., 2023), while Meta's production ServiceRouter shows that avoiding additional proxy hops at scale can save hundreds of thousands of machines (Saokar et al., 2023). As a capacity extrapolation, XSR's measured 6.8–37.2× throughput improvement would correspond to approximately 85–97% fewer routing instances if capacity scales proportionally with instance count. Repeated routing calls could accumulate these savings in agentic workloads, although application-level gains remain to be measured. Recent work shows that these repeated calls can accumulate latency and reduce resource efficiency (Luo et al., 2026; Chaudhry et al., 2026). Because semantic routing sits on the critical path for each request, its latency directly contributes to end-to-end latency. These observations motivate rethinking where semantic-routing work should run and what limits moving it closer to the datapath.

B Request-Path Sequence Comparison

Purpose and scope. The architecture comparison in the main text (Figure 2) emphasizes function placement, whereas this appendix makes the ordering and communication boundaries of the two request paths explicit. VSR supports additional signal types, decision logic, plugins, caching, and response processing that are omitted here because the comparison focuses on the request-side sequence through backend selection.

VSR reference. The VSR sequence is a simplified view based on the official Router API documentation and signal–decision architecture description (vLLM Semantic Router Team, 2026b; 2025), consistent with the high-level Envoy–ExtProc request path described by Haberman et al. (2025).

Critical-path interpretation. The figure illustrates communication structure rather than attributing measured overhead to individual components. VSR requires a per-request exchange between Envoy and the external router before forwarding to the selected backend. XSR instead performs request parsing, signal extraction, policy evaluation, and backend forwarding within the kernel socket datapath, avoiding this external-router exchange while preserving the required routing operations.



(a) VSR request flow. (b) XSR kernel-datapath request flow.

Figure 6: Request-path comparison of VSR and XSR. XSR removes the per-request Envoy–ExtProc exchange for supported routing functions saving three total hops.

C KEYWORD IMPLEMENTATION

XSR implements two keyword-routing functions in the datapath: character n-gram and BM25. For each, we describe the corresponding VSR behavior, the representations, and arithmetic used to preserve routing behavior within XSR's supported domain.

C.1 CHARACTER n-GRAM IMPLEMENTATION.

Matching behavior. For a character n-gram signal f_i, the extracted request input q_i is evaluated against the configured keyword set K_i. The signal parameters θ_i include the keyword set, similarity threshold τ_i, keyword operator, and case-sensitivity configuration. XSR preserves VSR's character n-gram matching behavior while replacing its userspace representation with bounded kernel state.

For single-word matching, the matcher evaluates individual words extracted from q_i; for phrase matching, it evaluates the complete request input q_i. We use x to denote the candidate string currently being compared with the configured keyword or phrase $k \in K_i$. Thus, x corresponds to an individual word for single-word matching and to the full request input for phrase matching. After applying the configured case normalization and boundary padding, both implementations construct character-trigram multisets $G_3(x)$ and $G_3(k)$. These trigrams consist of Unicode characters rather than UTF-8 bytes, distinguishing this representation from the byte-trigram features used by the distilled intent classifier. The following example illustrates these two matching scopes.

Example. Let the extracted request input be

$$q_i = \texttt{"Help me write Python code for quicksort."}.$$

Single-word matching: XSR and VSR evaluate the word candidates

$$W(q_i) = \{\texttt{Help}, \texttt{me}, \texttt{write}, \texttt{Python}, \texttt{code}, \texttt{for}, \texttt{quicksort}\}.$$

If the configured keyword is $k = \texttt{"coding"}$, then the matcher computes the n-gram similarity between k and each $x \in W(q_i)$, for example:

$$\text{score}(\texttt{code}, \texttt{coding}).$$

Phrase matching: The candidate set contains the complete request input:

$$P(q_i) = \{\texttt{"Help me write Python code for quicksort."}\}$$

Then for a configured phrase such as $k = \texttt{"write python"}$ the matcher evaluates:

$$\text{score}(q_i, k).$$

Similarity computation. After constructing $G_3(x)$ and $G_3(k)$, the matcher measures their overlap while preserving repeated trigrams. Let $c_x(g)$ and $c_k(g)$ denote the number of occurrences of trigram g in x and k, respectively. We define the shared trigram count as

$$o(x,k) = \sum_g \min(c_x(g), c_k(g)),$$

and the corresponding multiset union size as

$$u(x,k) = |G_3(x)| + |G_3(k)| - o(x,k).$$

VSR then applies the warp-2 similarity used by its character n-gram matcher:

$$\text{score}(x,k) = 1 - \left(\frac{u(x,k) - o(x,k)}{u(x,k)}\right)^2.$$

A candidate matches k when

$$\text{score}(x,k) \geq \tau_i.$$

The resulting keyword matches are combined using the configured OR, AND, or NOR operator to produce the binary signal s_i.

Integer threshold evaluation. The userspace implementation can evaluate the similarity expression directly using floating-point arithmetic. XSR instead uses integer arithmetic in the eBPF request path. XSR supports configured thresholds that are exactly representable to thousandth precision. For a supported threshold τ_i, XSR stores

$$T_i = 1000\tau_i.$$

Thresholds that cannot be represented exactly at this precision are rejected during policy generation. Substituting the score above, the comparison

$$\text{score}(x, k) \geq T_i/1000$$

can be evaluated without division or floating-point arithmetic as

$$1000\left(u^2 - (u-o)^2\right) \geq T_i u^2.$$

For example, a configured threshold of $\tau_i = 0.7$ corresponds to $T_i = 700$. This changes the arithmetic representation of the similarity test while preserving the same threshold decision for supported configurations.

Kernel representation. VSR's n-gram path operates on characters rather than raw UTF-8 bytes. XSR receives q_i as a byte stream, so applying the matcher directly to those bytes would change the trigram representation for multi-byte characters and could therefore change the resulting similarity score. XSR instead incrementally decodes the request stream into Unicode scalar values before constructing trigrams. Each trigram g is represented explicitly as three 32-bit Unicode scalar values, preserving the identity of the characters without introducing a hashed representation. For case-insensitive policies, ASCII characters are lowercased directly in the datapath, while additional one-to-one Unicode lowercase mappings required by the configured policy are precomputed in userspace and stored in a BPF map.

Configured keywords are also prepared before request processing. A userspace policy generator applies the same case normalization and boundary padding, constructs each keyword's distinct trigrams and their multiplicities $c_k(g)$, converts τ_i to its integer representation T_i, and records the associated rule metadata. The resulting state is then loaded into BPF maps for request-time evaluation.

During request processing, XSR maintains separate trigram state for the current word candidate and the complete-input candidate described above. As each trigram g is produced, XSR records how many times that trigram has appeared in the current candidate. A precomputed lookup indexed by the trigram and its occurrence number identifies which configured keywords should receive an additional overlap count. Thus, the j-th occurrence of g contributes to a keyword only when that keyword also contains at least j occurrences of g, implementing the multiplicity term

$$\min(c_x(g), c_k(g))$$

used in $o(x, k)$. Once a candidate is complete, its trigram count and accumulated overlap are used in the integer threshold comparison above. This representation allows XSR to reproduce VSR's character-level matching behavior using fixed kernel data structures and integer arithmetic.

Stage	VSR userspace	XSR datapath	Preserved behavior
1. Normalize	Rust string processing	Limited Unicode decoding and precomputed case mappings	Case normalization and character-level representation
2. Trigrams	`ngrammatic`	Explicit character trigrams with precomputed BPF-map state	Boundary-padded character trigrams
3. Overlap	Multiset multiplicity	Per-trigram occurrence counters and bounded lookups	Multiplicity-aware overlap
4. Score	Floating-point warp-2 score	Integer-equivalent threshold evaluation	Same inclusive similarity decision
5. Rule	Threshold and keyword operator	Fixed-width signal masks and policy entries	Same AND/OR/NOR rule outcome

Table 2: Translation of VSR's character-trigram matching stages to XSR's datapath implementation. The implementations use different representations and operations while preserving the corresponding matching behavior within XSR's supported input domain.

C.2 BM25 IMPLEMENTATION.

Matching behavior. For a BM25 signal f_i, VSR treats each configured keyword or phrase $k \in K_i$ as a document and the extracted request input q_i as the query. Unlike character n-gram matching, BM25 evaluates the complete request input rather than separately evaluating word- and phrase-level candidates. Both the keyword documents and query are first tokenized before their relevance score is computed.

Let $\mathcal{T}(x)$ denote the token sequence produced from text x, and let $N_i = |K_i|$ denote the number of configured keyword documents. For token t, let $\mathrm{tf}(t,k)$ denote its frequency in keyword document k, and let

$$\mathrm{df}_i(t) = |\{k \in K_i : t \in \mathcal{T}(k)\}|$$

denote the number of keyword documents containing t. The average keyword-document length is

$$\mathrm{avgdl}_i = \frac{1}{N_i} \sum_{k \in K_i} |\mathcal{T}(k)|.$$

Following VSR's BM25 configuration, the contribution of token t to keyword document k is

$$w_i(t,k) = \log\left(1 + \frac{N_i - \mathrm{df}_i(t) + 0.5}{\mathrm{df}_i(t) + 0.5}\right) \cdot \frac{\mathrm{tf}(t,k)(k_1+1)}{\mathrm{tf}(t,k) + k_1\left(1 - b + b\frac{|\mathcal{T}(k)|}{\mathrm{avgdl}_i}\right)},$$

where $k_1 = 1.2$ and $b = 0.75$. The BM25 score for k is then

$$\mathrm{BM25}(q_i,k) = \sum_{t \in \mathcal{T}(q_i)} w_i(t,k),$$

where repeated query tokens contribute repeatedly. A keyword document matches when

$$\mathrm{BM25}(q_i,k) \geq \tau_i,$$

and the resulting document matches are combined using the configured OR, AND, or NOR operator to produce s_i.

Precomputed score representation. The document frequency, term frequency, document length, and inverse-document-frequency terms above depend only on the configured keyword corpus K_i. XSR therefore computes these values before request processing rather than reproducing the complete BM25 calculation in eBPF. For each token t and keyword document k, the userspace policy generator computes $w_i(t,k)$ and stores a scaled integer weight

$$W_i(t,k) = \mathrm{round}\left(10^6 w_i(t,k)\right).$$

The configured threshold is represented using the same scale,

$$\widehat{\tau}_i = \mathrm{round}\left(10^6 \tau_i\right).$$

Request-time scoring therefore reduces to

$$\widehat{\mathrm{BM25}}(q_i,k) = \sum_{t \in \mathcal{T}(q_i)} W_i(t,k),$$

followed by the integer comparison

$$\widehat{\mathrm{BM25}}(q_i,k) \geq \widehat{\tau}_i.$$

This moves the logarithm, document-frequency calculation, and length normalization out of the request path, leaving only integer accumulation and comparison in eBPF.

Kernel representation. VSR's BM25 tokenizer performs text normalization, stop-word removal, and English stemming before scoring. These userspace tokenizer operations cannot be directly reused in eBPF, so XSR separates the static and request-dependent portions of token processing. Within the supported BM25 domain, configured keyword documents are restricted to ASCII text and are normalized, tokenized, stop-word filtered, and Porter2-stemmed by the userspace policy generator.

XSR avoids executing the Porter2 stemmer in the request path by expanding each configured stem into a bounded set of supported surface forms during policy generation. Surface forms that reduce to the same stem are assigned the same precomputed document-weight vector. At request time, XSR lowercases and incrementally tokenizes q_i, computes a 32-bit FNV-1a hash for each token, and uses that hash to locate its precomputed weight vector in a BPF map. Map entries also retain bounded token information that is checked before the corresponding weights or stop-word behavior are applied. Stop words are represented in the same map with zero document weights and are discarded before scoring. The policy generator rejects conflicting generated token entries during policy construction.

For each accepted query token, the corresponding weight is added to the running score of every configured keyword document. Repeated query tokens therefore add the same weight repeatedly, preserving their contribution to the VSR score. Once a document's accumulated score reaches its scaled threshold, XSR marks that document as matched. The resulting document mask is then evaluated using the configured keyword operator to produce the BM25 signal s_i.

Evaluation policy. The character n-gram and BM25 evaluations use the same four keyword groups for coding, math, QA, and writing, with route priority coding $>$ math $>$ QA $>$ writing and *others* as the fallback. The corresponding keyword sets are {`code`, `function`, `program`, `implement`}, {`solve`, `equation`, `derivative`, `calculate`}, {`question`, `answer`, `explain`, `define`}, and {`write`, `essay`, `draft`, `compose`}, respectively. Character n-gram uses $n = 3$ with threshold 0.8, while BM25 uses threshold 0.1; both use case-insensitive OR matching.

Stage	VSR userspace	XSR datapath	Matching behavior
1. Normalize	Text normalization and lowercase processing	Request-time lowercase processing with policy-time normalization	Case-insensitive token representation
2. Tokenize	Tokenization, stop-word removal, and Porter2 stemming	Incremental tokenization with precomputed surface forms and stop-word entries	Supported token mapping
3. Weight	Runtime BM25 term weighting using TF, DF, IDF, and length normalization	Precomputed per-token document weights stored in BPF maps	Scaled configured document weights
4. Score	Floating-point accumulation of BM25 scores	Integer accumulation of scaled precomputed weights	Inclusive scaled-threshold comparison
5. Rule	Threshold and keyword operator	Fixed-width document masks and policy entries	Same AND/OR/NOR rule evaluation

Table 3: Translation of VSR's BM25 matching stages to XSR's datapath implementation. XSR precomputes document-dependent BM25 terms and reimplements request-time scoring with bounded integer operations over its supported input domain (Appendix F).

D INTENT CLASSIFIER DISTILLATION

Datapath feasibility. Model-based signals require substantially more state and computation than the deterministic keyword functions considered above. VSR's mmBERT-32K intent classifier, for example, requires approximately 1.2 GB of parameter storage. Although eBPF programs have only 512 bytes of stack space (The Linux Kernel Documentation, 2026), this restriction alone does not preclude learned inference: prior work has demonstrated lightweight quantized neural-network inference in eBPF using integer-only arithmetic (Hara & Sasabe, 2024).

The primary obstacle for directly executing mmBERT-32K is therefore the state and computation required by transformer inference under eBPF's bounded execution environment. XSR instead performs mmBERT inference offline and distills its predictions into the kernel-compatible student described below.

Teacher model. XSR uses VSR's published mmBERT-32K intent classifier as an offline teacher (vLLM Semantic Router, 2026). The teacher consists of the `mmbert-32k-yarn` base sequence classifier with the published intent-classification LoRA adapter applied on top (vLLM Semantic Router Team, 2026a). Both the base model and adapter are frozen; XSR does not retrain either component. Given request input q_i, the teacher produces a logit vector

$$\ell(q_i) \in \mathbb{R}^{14}$$

over the 14 intent domains. These logits provide the soft training targets for the student and are generated offline before student training. The class ordering is fixed across teacher-target generation, student training, model export, and kernel inference.

Training data. Student training uses labeled prompts drawn from MMLU-Pro (Wang et al., 2024) and the VSR category-classifier supplement. Prompts are deduplicated using the SHA-256 digest of the normalized byte representation used by the student. During development, we retain a fixed 72-prompt validation set and a 343-prompt holdout while expanding the training set to 4,900 prompts, with 350 examples for each of the 14 domains. The additional training examples are deterministically sampled from unused MMLU-Pro rows while leaving the fixed validation and holdout sets unchanged. Because some MMLU-Pro official-test rows are explicitly reassigned to student training during this expansion, the 343-prompt set is treated only as a fixed development holdout rather than as evaluation over the full official MMLU-Pro test split.

Evaluation data. The independent intent evaluation uses an internally constructed 1,400-request corpus, `xsr-routing-v1`, balanced across the 14 domains with 100 requests per domain. It contains 50 topic clusters per domain with two prompts per topic. Normalized prompts were checked against the complete MMLU-Pro and VSR category-classifier-supplement manifests used for student training and development, with no exact overlaps or near duplicates at the 0.9 similarity threshold. This establishes independence from the documented student-training sources, but not from undocumented teacher pretraining data.

Student representation. The student uses a representation designed to map directly to the operations used during kernel inference. Let

$$b(q_i) = (b_1, \ldots, b_L)$$

denote the UTF-8 byte sequence of q_i, truncated to at most 16,384 bytes after converting only ASCII `A-Z` bytes to lowercase. All other bytes are left unchanged. The student constructs the multiset of consecutive byte trigrams

$$T(q_i) = \{(b_j, b_{j+1}, b_{j+2})\}_{j=1}^{L-2}.$$

Each trigram t is hashed using 32-bit FNV-1a into one of 8,192 feature buckets,

$$h(t) = \mathrm{FNV1a}_{32}(t) \bmod 8192.$$

Unlike the character-n-gram keyword implementation in Appendix C, this representation operates directly on bytes and does not perform Unicode character decoding.

The student is a linear 14-domain classifier. For domain c, its score is

$$z_c(q_i) = \beta_c + \sum_{t \in T(q_i)} W_{c,h(t)},$$

where $W \in \mathbb{R}^{14\times 8192}$ contains the learned feature weights and β_c is the class bias. Repeated trigrams contribute repeatedly to the sum. The predicted domain is

$$\hat{c} = \arg\max_c z_c(q_i).$$

Distillation objective. For each training request, the student is optimized using both its ground-truth domain label y and the frozen teacher logits $\ell(q_i)$. Let $z(q_i)$ denote the student logits. The distilled training objective is

$$\mathcal{L} = \alpha\ \mathrm{CE}(y, z(q_i)) + (1-\alpha)T^2\,\mathrm{KL}\left(\mathrm{softmax}\left(\frac{\ell(q_i)}{T}\right) \middle\| \mathrm{softmax}\left(\frac{z(q_i)}{T}\right)\right).$$

An initial sweep considers $T \in \{1, 2, 4\}$ and $\alpha \in \{0.25, 0.5\}$, selecting checkpoints first by validation teacher agreement and then by validation ground-truth accuracy. The selected training configuration uses $T = 2$ and $\alpha = 0.25$. The final student is trained for 12 epochs using balanced minibatch sampling with batch size 128 and AdamW with learning rate 0.02 and weight decay 10^{-4}.

Student model selection. After expanding the balanced training set to 4,900 prompts, we evaluate whether feature-space width or additional model capacity improves the student. We compare 4K, 8K, and 16K hash widths and select 8K as the smallest representation improving both held-out teacher agreement and accuracy over 4K; the distilled 16K model does not improve further. A 16-unit nonlinear alternative also provides insufficient benefit, so the deployed student remains a linear classifier with 8,192 hashed byte-trigram features.

Quantized kernel representation. After model selection, XSR converts the floating-point student to the integer representation used during request processing. A single global symmetric scale is applied to W, producing INT8 weights $\widehat{W} \in \mathbb{Z}^{14\times 8192}$. The biases are multiplied by the same scale and rounded to INT32 values $\widehat{\beta}_c$. Kernel inference therefore computes

$$\widehat{z}_c(q_i) = \widehat{\beta}_c + \sum_{t \in T(q_i)} \widehat{W}_{c,h(t)}$$

using integer accumulation, followed by an integer argmax over the 14 scores.

The exported model contains 114,688 INT8 weights and 14 INT32 biases. The weights are loaded into an 8,192-entry BPF array map, with each entry containing the 14 class weights associated with one hash bucket. During request processing, each byte trigram therefore requires one map lookup followed by 14 score updates. The exporter also derives an upper bound on the accumulated INT32 scores from the 16,384-byte prompt limit and rejects a model if this bound exceeds the signed 32-bit range. The userspace student reference and the XSR kernel implementation use the same normalization, FNV-1a hashing, weight representation, bias values, and argmax rule.

Intent quality of the deployed student. We evaluate the INT8 student on the independent 1,400-request labeled test set used in Section 3.2.

Table 1 compares its classification quality with the mmBERT-32K teacher. In addition to 14-domain classification, we map both predictions and ground-truth labels through the shared routing policy, where computer science maps to *coding*, math maps to *math*, and the remaining 12 domains map to *others*.

The student retains substantially more routing-level accuracy than domain-level accuracy because 12 of the 14 domains map to the *others* route. Since this route contains 1,200 of the 1,400 test requests, route accuracy is dominated by the majority route; macro-F1 provides a more balanced measure. The deployed student therefore demonstrates kernel-compatible learned inference, but does not preserve the full classification quality of the mmBERT-32K teacher.

E EXPERIMENTAL ENVIRONMENT

Hardware. All experiments were conducted on a Microsoft Azure `Standard_D4as_v4` virtual machine with 4 vCPUs and 16 GiB of memory. The VM ran Ubuntu 24.04 LTS with Linux kernel `6.17.0-1022-azure`. XSR, VSR, LLMRouter (LLMR), the workload generator, and the marker backends shared this Azure VM and therefore the same underlying compute allocation. XSR executed directly against the VM's Linux kernel, whereas the VSR baseline ran through its containerized Envoy and semantic-router components.

Kernel and eBPF requirements. XSR uses `BPF_PROG_TYPE_SK_SKB` stream parsing and `BPF_MAP_TYPE_SOCKMAP` redirection after TLS termination. The implementation also uses eBPF loops and therefore depends on compatible helper and verifier behavior. The evaluated Azure kernel had `CONFIG_BPF`, `CONFIG_BPF_SYSCALL`, and `CONFIG_BPF_STREAM_PARSER` enabled. Runtime feature probing additionally confirmed support for `SK_SKB` programs and `SOCKMAP` maps. Deployment therefore assumes TLS termination before XSR, plaintext HTTP/1.1 visibility at the socket datapath, and a Linux kernel with equivalent socket-layer eBPF support. Exact configuration may vary across distributions and kernel releases. Vieira et al. (2020) provide general background on eBPF; current kernel documentation is available at `https://docs.kernel.org/bpf/` and `https://docs.kernel.org/bpf/map_sockmap.html`.

Network and container topology. The workload generator ran in a separate Linux network namespace connected to the Azure VM's host namespace through a virtual Ethernet pair. The direct-backend control, XSR, and LLMR traversed this host-veth path. XSR's userspace control process and marker backends ran in the host environment, while its `SK_SKB` programs executed in the VM kernel. XSR accepted the plaintext HTTP stream in the host namespace and redirected requests to the selected backend sockets through `SOCKMAP`. LLMR ran as a userspace service in the host environment and forwarded requests to the same marker backends.

VSR used the containerized vLLM Semantic Router v0.3.0 deployment. Envoy and the external semantic-router process ran in separate Docker containers connected through a Docker bridge, with Envoy invoking the router through ExtProc before forwarding to the selected marker backend. The Envoy-only control used the same Docker environment with ExtProc disabled to isolate Envoy and container overhead. A `vllm-sr-sim` sidecar was also launched but remained idle and was not part of the measured request path.

A Docker-bridge XSR mode was not used because `SOCKMAP` entries refer to sockets opened within the relevant network namespace. Moving only the classifier into a privileged container would not reproduce the same frontend and backend socket ownership and would therefore not provide a valid topology-matched path.

Control-plane state. A userspace control process establishes backend connections and installs the socket, route, and policy mappings consumed by the XSR kernel datapath. The evaluated prototype uses static backend mappings after initialization. It does not yet incorporate continuously changing backend health, autoscaling events, or model availability, although these values could be supplied through map updates in a future control plane.

Evaluated request paths. Table 4 summarizes the request paths and controls used in the performance comparison.

Path	Request processing	Topology
Direct backend	No proxy or semantic routing	Host veth
Envoy only	Envoy forwarding without ExtProc	Docker bridge
XSR	`SK_SKB` parsing, signal and policy evaluation, and `SOCKMAP` redirect	Host veth
VSR	Envoy forwarding with ExtProc and the external router	Docker bridge
LLMR	Userspace semantic routing and backend forwarding	Host veth

Table 4: Evaluated request paths and controls.

F SUPPORTED INPUT DOMAIN

Preserved keyword-routing semantics. For character n-gram routing, XSR preserves VSR's case normalization, boundary-padded character trigrams, trigram multiplicities, warp-2 scoring, inclusive threshold comparison, keyword operators, route priorities, and backend mappings. For BM25, XSR implements the configured keyword-document scoring and rule evaluation over the bounded token transformation described below. Supported input bounds do not guarantee identical routing decisions to VSR: Section 3.2 reports four disagreements in 9,280 requests, with XSR matching an independent BM25 policy reference in those four cases.

Prototype bounds. The current character n-gram implementation supports trigrams ($n = 3$), with at most 16 keywords and 8 rules, 32 trigrams per word or keyword, and 128 trigrams for full-text phrase matching. Character n-gram thresholds must be exactly representable to thousandth precision; unsupported threshold values are rejected during policy generation. The BM25 implementation supports at most 16 configured keyword documents and 8 rules; configured BM25 keyword documents are restricted to ASCII text and are normalized and tokenized during policy generation. Generated BM25 terms are bounded to 64 bytes, with bounded token state retained for request-time lookup validation. Request processing is bounded to 2,000-byte HTTP headers and 256 KiB complete requests. The evaluated request format contains a single string-valued user `content` field.

The character n-gram implementation supports bounded Unicode code-point processing, raw UTF-8 and JSON Unicode escapes, and both per-word and multi-word phrase matching. Case-insensitive policies support bounded one-to-one Unicode lowercase mappings, while expanding or context-sensitive case transformations remain unsupported. For BM25, unsupported policy configurations are rejected during policy generation. Request text may contain UTF-8, but XSR treats each non-ASCII code point as a word boundary rather than reproducing VSR's Unicode transliteration. Within ASCII words, it follows VSR's normalization and stemming order.

Model-based signal bounds. The evaluated intent signal uses the distilled 14-domain student described in Appendix D. Request content is truncated to 16,384 UTF-8 bytes, ASCII uppercase bytes are lowercased, and byte trigrams are hashed into 8,192 feature buckets. The deployed student uses 14 output classes with INT8 weights and INT32 accumulation. These bounds apply to the evaluated distilled classifier and do not imply that arbitrary learned models can execute in the datapath.

Request framing and parsing. The evaluated request domain uses plaintext HTTP/1.1 requests with CRLF-terminated headers, explicit `Content-Length` framing, and an OpenAI-style Chat Completions body containing a single string-valued user `content` field. The parser requires the literal key sequence `"content":`, with no whitespace between the closing quote and the colon. Only spaces or tabs may appear between the colon and the opening quote of the content value. Line breaks at this position are unsupported. Within these bounds, the parser supports raw UTF-8 and JSON Unicode escapes. Chunked transfer encoding, malformed or missing length framing, unsupported JSON layouts, oversized requests, and HTTP pipelining fall outside the supported domain. The current prototype provides no application-level fallback or error handling for these cases.

Unsupported routing functions. Embedding-based signals and full transformer-based classifiers such as mmBERT-32K require computation beyond what XSR currently supports and therefore remain in the application layer. XSR supports the evaluated distilled intent classifier because its representation and inference have been reimplemented to fit within eBPF's execution constraints. Other learned routing functions are not supported unless they admit a compatible datapath implementation. The current prototype also does not provide automatic fallback when a policy contains an unsupported function; evaluated policies must therefore use supported routing functions.

G Benchmark Protocol and Constraints

Correctness evaluation. Keyword-routing correctness uses a combined 9,280-request corpus consisting of 880 requests from the qualitative test split of NVIDIA SPEED-Bench (Abramovich et al., 2026) and the full 8,400-request RouteWorks RouterArena corpus (Lu et al., 2026). The keyword corpus is used only for routing-correctness evaluation and is not used to train the intent student.

Throughput and latency workloads. The throughput experiment uses standard `wrk` to measure throughput as request concurrency increases. Latency is evaluated separately at a fixed request rate of 100 req/s so that compared systems receive the same request rate. HTTP keep-alive is enabled, and each system receives the same sequence of request bodies. Timed runs do not parse every response body; the correctness benchmark and untimed preflight requests check route behavior, while runs containing transport, timeout, or HTTP errors are rejected.

Repeated trials and reporting. The benchmark profile uses five independent 40-second trials, each preceded by a 3-second warm-up. XSR uses the same SOCKMAP-router process for warm-up and measurement within each trial. For throughput runs, measurement begins after closed warm-up connections and their map state have been reclaimed. Fixed-rate latency runs reuse the same four persistent HTTP connections across both phases. The router is stopped after measurement and started again for the next trial. System order varies across trials, and raw outputs are retained. Throughput and average latency are reported as five-trial means with 95% confidence intervals. P50, P95, and P99 latency are averaged across the per-trial percentiles.

Marker backends and interpretation. The lightweight marker backends parse each request and return a fixed response identifying the selected route. This design minimizes backend computation so that the measurements primarily reflect parsing, signal evaluation, route selection, proxy communication, and forwarding. The resulting values characterize routing-path overhead, not backend LLM inference latency, generated-token throughput, time to first token, or application-level response quality.

Shared virtualized host. All evaluated components share the same Azure VM and therefore compete for its CPU and memory resources. This controlled setup reduces hardware variation but does not reproduce every property of a bare-metal or distributed serving deployment. In addition, XSR, VSR, and LLMRouter do not all execute through identical host and container paths, so the complete-path comparisons should not be interpreted as attributing every measured difference to a single kernel or userspace operation.

Concurrency bound. The benchmark sweep is capped at 512 simultaneous connections, while the main figures report only the common valid concurrency ranges for each comparison. Each synthetic backend uses a pool of 512 worker threads, so higher concurrency could make backend worker availability and queueing dominate the routing measurement. Evaluating larger connection counts would require a backend whose independently measured capacity exceeds every routed configuration.

Benchmark resource limits. High-concurrency runs require sufficient file descriptors for the workload generator, routers, backend listeners, and active connections. The benchmark scripts check and raise the effective file-descriptor limit before measurement and record it in the run metadata.

Corpus	N	Mean	P50	P95	P99	Max
SPEED-Bench	880	664.66	121	3232.05	8366.16	14166
RouterArena	8400	726.81	286	2591.20	4809.22	10008
Combined keyword	9280	720.91	269	2776.10	4945.78	14166
Intent evaluation	1400	59.44	60	70	73	78

Table 5: UTF-8 prompt-length distributions, in bytes, for the correctness workloads.

All prompts in Table 5 fall within the corresponding XSR request-size bounds described in Appendix F.

H Detailed Experimental Results

This appendix reports the numerical values underlying Figures 4 and 5. The throughput and XSR/VSR fixed-rate latency comparisons use five valid trials. LLMR latency at 100 req/s is retained only as an overload diagnostic, not as a valid steady-state comparison. Throughput and average-latency values are reported as the trial mean ± the 95% confidence-interval margin. Latency percentiles report the mean of the per-trial P50, P95, and P99 values.

H.1 Routing Correctness

Signal	Comparison	Agreement
N-gram	XSR–VSR	9,280/9,280 (100.00%)
N-gram	XSR–LLMR	9,280/9,280 (100.00%)
BM25	XSR–VSR	9,276/9,280 (99.96%)
BM25	XSR–LLMR	9,280/9,280 (100.00%)
Intent domain	Student–Teacher	417/1,400 (29.79%)
Intent route	Student–Teacher†	1,240/1,400 (88.57%)
Intent route	XSR–Envoy+ExtProc*	1,400/1,400 (100.00%)
Intent route	XSR–LLMR	1,400/1,400 (100.00%)

Table 6: Agreement across routing signals and implementations. *Envoy+ExtProc uses the same student model as XSR. †Student–Teacher routing agreement is derived under the shared routing policy.

H.2 Throughput across concurrency levels

Tables 7–9 report throughput across the evaluated concurrency levels shown in Figure 4.

c	XSR (req/s)	VSR (req/s)	LLMR (req/s)
1	1993.194 ± 8.020	295.028 ± 0.371	30.646 ± 0.057
2	3578.698 ± 28.092	400.364 ± 5.161	29.624 ± 0.023
4	6026.160 ± 20.095	464.366 ± 5.754	34.434 ± 0.166
8	6075.648 ± 41.626	507.064 ± 5.700	34.684 ± 0.112
16	6041.870 ± 27.468	532.210 ± 5.074	34.754 ± 0.005
32	6036.912 ± 7.662	543.028 ± 3.141	34.350 ± 0.000
64	6007.190 ± 30.538	540.692 ± 5.112	33.544 ± 0.005
96	5963.006 ± 44.596	537.096 ± 10.155	33.550 ± 0.000
128	5975.040 ± 31.019	542.892 ± 17.594	31.942 ± 0.004

Table 7: Character n-gram throughput across the evaluated concurrency levels. Values are five-trial means ± 95% confidence-interval margins.

c	XSR (req/s)	VSR (req/s)	LLMR (req/s)
1	5673.730 ± 13.217	535.120 ± 2.210	44.184 ± 0.131
2	10070.820 ± 36.973	900.154 ± 32.852	44.128 ± 0.099
4	22518.008 ± 73.281	1285.540 ± 5.745	43.392 ± 0.192
8	22867.260 ± 134.533	1539.234 ± 7.975	44.504 ± 0.265
16	22784.830 ± 57.262	1692.806 ± 8.372	45.116 ± 0.254
32	22716.184 ± 97.067	1781.692 ± 18.290	43.976 ± 0.076
64	22595.420 ± 65.316	1844.494 ± 22.691	43.128 ± 0.004
96	22509.030 ± 100.004	1866.284 ± 23.081	43.134 ± 0.005
128	22281.650 ± 122.728	1894.458 ± 9.591	41.526 ± 0.008

Table 8: BM25 throughput across the evaluated concurrency levels. Values are five-trial means ± 95% confidence-interval margins.

c	XSR (req/s)	VSR* (req/s)	LLMR (req/s)
1	8808.150 ± 83.049	926.518 ± 2.967	48.252 ± 0.670
2	15176.200 ± 50.330	1163.218 ± 9.757	47.470 ± 0.318
4	39397.468 ± 437.366	1117.066 ± 4.534	47.408 ± 0.961
8	41134.686 ± 549.501	1104.722 ± 3.903	48.360 ± 0.390
16	41286.820 ± 160.067	1114.640 ± 2.142	48.410 ± 0.376

Table 9: Intent throughput across the evaluated concurrency levels. VSR* denotes the Envoy+ExtProc path executing the same student model as XSR. Values are five-trial means ± 95% confidence-interval margins.

H.3 Fixed request rate latency

Tables 10–12 report latency at a fixed request rate of 100 req/s. Average latency is reported as the five-trial mean ± the 95% confidence-interval margin, while P50, P95, and P99 report means of the corresponding per-trial percentiles. LLMR results are retained here and in Figure 5 only as overload diagnostics: †LLMR did not sustain the offered 100 req/s, so these values are not valid steady-state latency comparisons. Reported latency reductions compare XSR with VSR only.

RPS	System	Avg. ± 95% CI (ms)	P50	P95	P99
100	XSR	0.809 ± 0.003	0.538	2.308	4.964
	VSR	3.124 ± 0.013	2.277	8.418	14.532
	LLMR†	135.715 ± 0.870	99.619	344.128	427.924

Table 10: Character n-gram latency at 100 RPS. Latency values are in milliseconds. Averages show the five-trial mean and 95% confidence-interval margin; percentiles are means of per-trial percentiles.

RPS	System	Avg. ± 95% CI (ms)	P50	P95	P99
100	XSR	0.558 ± 0.014	0.507	0.881	1.490
	VSR	2.749 ± 0.037	2.572	4.081	5.502
	LLMR†	91.412 ± 0.319	84.860	126.219	148.351

Table 11: BM25 latency at 100 RPS. Latency values are in milliseconds. Averages show the five-trial mean and 95% confidence-interval margin; percentiles are means of per-trial percentiles.

RPS	System	Avg. ± 95% CI (ms)	P50	P95	P99
100	XSR	0.389 ± 0.002	0.387	0.493	0.552
	VSR*	1.445 ± 0.018	1.424	1.682	1.920
	LLMR†	81.943 ± 0.871	81.471	84.761	87.766

Table 12: Intent latency at 100 RPS. VSR* is the Envoy+ExtProc path running the same student model as XSR. Latency values are in milliseconds. Averages show the five-trial mean and 95% confidence-interval margin; percentiles are means of per-trial percentiles.