Efficient Multi-Model Orchestration for Self-Hosted Large Language Models

Bhanu Prakash Vangala[1], **Tanu Malik**[2]

[1]Department of Electrical Engineering and Computer Science
University of Missouri, Columbia
Columbia, MO 65211, USA
bv3hz@missouri.edu, tanu@missouri.edu

Abstract

Self-hosting large language models (LLMs) is increasingly appealing for organizations seeking privacy, cost control, and customization. Yet deploying and maintaining in-house models poses challenges in GPU utilization, workload routing, and reliability. We introduce Pick and Spin, a practical framework that makes self-hosted LLM orchestration scalable and economical. Built on Kubernetes, it integrates a unified Helm-based deployment system, adaptive scale-to-zero automation, and a hybrid routing module that balances cost, latency, and accuracy using both keyword heuristics and a lightweight DistilBERT classifier. We evaluate four models Llama-3 (90B), Gemma-3 (27B), Qwen-3 (235B), and DeepSeek-R1 (685B) across eight public benchmark datasets, with five inference strategies, and two routing variants encompassing 31,019 prompts and 163,720 inference runs. Pick and Spin achieves up to 21.6% higher success rates, 30% lower latency, and 33% lower GPU cost per query compared with static deployments. The results show that intelligent orchestration and efficient scaling enable enterprise-grade LLM performance on self-hosted infrastructure, bringing high-capacity AI within affordable reach.

Introduction

Large language models (LLMs) are rapidly transforming applications across domains. Instead of relying on a single general purpose model for all tasks, both research and industry are moving toward a diverse ecosystem of domain tuned models. These specialized models, trained for fields such as scientific research, finance, law, and healthcare, provide greater precision and contextual understanding within their respective areas. However, this specialization also introduces new challenges. Organizations must decide not only which model to use between general purpose and fine tuned variants but also how to deploy and manage them efficiently while maintaining data privacy and minimizing computational cost.

This challenge has led to what can be described as the self hosting dilemma. On one side, relying on commercial APIs from providers such as OpenAI, Gemini, or Claude simplifies deployment but introduces vendor lock in, unpredictable costs, and data exposure risks that are unacceptable in sensitive domains such as healthcare, finance, or the life sciences (AI 2023). On the other side, self hosting preserves privacy and institutional control but comes with operational burdens. Static, always on deployments keep GPUs active even when idle, wasting resources and increasing energy consumption and maintenance overhead (Nguyen 2025).

Although domain tuned or distilled large language models (DT-LLMs) improve efficiency for specific use cases, they remain optimized for narrow objectives and cannot generalize to all query types. In practical applications, prompts vary widely, some require reasoning, others summarization or factual recall. No single model performs best across all these dimensions in there respective domains. For example, a model fine tuned for reasoning may perform poorly on summarization or fact retrieval. This diversity creates a key challenge in managing multiple models so that each input is served by the most suitable one without wasting computational resources or increasing latency.

The open source ecosystem, enabled by initiatives such as LLaMA and OPT, has made high quality model weights widely available. However, efficient and affordable deployment of these models remains difficult. Each model behaves differently across tasks, and their varying computational requirements make selection and scheduling complex. Organizations must balance accuracy, responsiveness, and cost, particularly when operating within private or resource constrained environments. These challenges motivate the need for an automated system that can both select the right model for each prompt and allocate resources intelligently

In this context, orchestration refers to the automated coordination of models and computational resources. It involves deciding which model to invoke, when to start or stop it, and how to allocate GPUs efficiently. Prior work in distributed systems defines orchestration as the automation and optimization of workflows to ensure scalability and reliability (Burns et al. 2016; Verma et al. 2015). Extending this principle to multi model inference enables a balance between accuracy, latency, and cost. Instead of keeping all models continuously active, the orchestrator routes simple queries to lightweight models and reserves larger ones for complex tasks. Idle models are scaled to zero, ensuring that GPU resources are used only when needed.

Existing research on model serving (Crankshaw et al. 2017), autoscaling (Baylor et al. 2017; Nguyen 2025), and serverless inference (Yu et al. 2022; Wang 2024) focuses

on specific parts of the orchestration pipeline but does not provide an integrated solution. These systems improve efficiency within individual layers such as inference scheduling or container scaling, yet they do not coordinate model selection and resource allocation together. Tools like Helm (Contributors 2019) and Knative extend Kubernetes with declarative configuration and event driven scaling, offering strong primitives for deployment automation. However, they lack mechanisms for task aware routing or model level coordination, which are essential for managing multiple language models under shared infrastructure.

To address this gap, we propose Pick and Spin (PS), a multi model orchestration framework that integrates intelligent routing with orchestration aware scaling. The system's name encapsulates its dual nature: Pick represents the intelligent routing layer that selects optimal models based on prompt complexity, while Spin represents the dynamic orchestration layer that manages model lifecycles, spinning resources up on demand and down when idle. PS formulates orchestration as a joint optimization problem balancing three objectives: model relevance, latency, and cost. A lightweight routing layer selects the best model for each query using rule based and semantic (DistilBERT) classifiers that estimate prompt complexity and intent. The orchestration layer manages model activation and deactivation using Kubernetes based scaling policies, ensuring efficient GPU utilization and minimal cold start delay.

PS is implemented on Kubernetes with Helm based control, using a unified umbrella chart that automates deployment, versioning, and recovery across multiple models and backends such as vLLM, TensorRT LLM, and TGI. This design enables reproducible deployment, zero downtime upgrades, and automatic fault recovery while keeping resource usage cost efficient. By combining content aware routing, dynamic scaling, and fault tolerant orchestration, Pick and Spin transforms static, resource heavy model hosting into a scalable and adaptive workflow. It provides a foundation for private multi model AI deployments that maintain performance, privacy, and cost efficiency within institutional infrastructure.

Related Work

Pick and Spin (PS) connects three major research directions: efficient LLM inference, multi model orchestration, and serverless computing. Early work such as Megatron LM (Shoeybi et al. 2019) explored large scale parallelism for training and serving billion parameter models. Modern systems like Text Generation Inference (TGI) (Team 2024), vLLM (Kwon et al. 2023), and DeepSpeed Inference (Aminabadi 2022) focus on runtime efficiency through memory optimization and parallel scheduling. For instance, the PagedAttention mechanism in vLLM manages the KV cache efficiently to reduce fragmentation and increase throughput. FastServe (Yu 2023) and SGLang (Zheng 2024) further streamline inference pipelines but target single model deployments, lacking cross model orchestration.

The increasing diversity of models has renewed interest in dynamic routing. Conceptually, this aligns with the Mixture of Experts (MoE) paradigm where a controller routes tokens to specialized submodules within a single network (Kirakosyan 2025; Pandit 2023). PS generalizes this concept to the system level by routing full prompts between independently deployed models. Recent systems such as UniRoute (Jitkrittum et al. 2025) and ModelSAT (Zhang, Zhan, and Ye 2025) use embeddings or instruction tuned heuristics to predict the best model efficiently, while controller based orchestration frameworks (Xie et al. 2025) leverage larger models for coordination but at higher latency and cost. PS takes a middle ground by using a DistilBERT-based classifier that balances semantic precision and efficiency.

Serverless architectures provide a natural fit for adaptive model deployment. Event driven and scale-to-zero execution models reduce idle GPU usage (Li and Yin 2024), but applying them to LLM workloads introduces challenges such as cold start latency (Satzke et al. 2022). Kubernetes based frameworks like Knative and KEDA enable autoscaling and asynchronous processing for AI services. InferLine (Crankshaw 2020) and ModelSwitch (Li 2022) propose latency aware scheduling and dynamic resource adaptation but focus on homogeneous pipelines. PS builds on these ideas by integrating Knative for low latency inference with KEDA for background scaling, achieving responsiveness and efficiency under fluctuating demand.

Self-hosted orchestration also contributes to responsible and accessible AI. Keeping inference within organizational boundaries supports compliance with regulations such as GDPR and HIPAA while safeguarding sensitive data (Khezresmaeilzadeh et al. 2025; Feretzakis et al. 2024). By combining open source tools with orchestration efficiency, PS enables smaller organizations and research institutions to deploy multi-model AI systems that were previously limited to large enterprises. These prior efforts provide the foundation for Pick and Spin, which unifies inference optimization, semantic routing, and orchestration aware scaling into a single adaptive framework.

Multi Model Orchestration Problem

We formulate the multi model orchestration problem as a joint optimization task that balances model relevance, latency, and cost. The goal is to automatically route each query to the most suitable model while ensuring efficient GPU utilization and minimal overhead.

Let $\mathcal{L}$ denote the set of available language models and $\mathcal{I}$ the set of inference backends. Each deployable combination (L_x, I_y) forms a service instance $S_{x,y}$ that can handle a query. For every prompt p, the system selects the instance (x^*, y^*) that maximizes performance while minimizing latency and cost:

$$(x^*, y^*) = \arg\max_{(x,y)} f(p, S_{x,y}) \tag{1}$$

where the scoring function is defined as

$$f(p, S_{x,y}) = \alpha R(p, L_x) - \lambda T(S_{x,y}) - \mu C(S_{x,y}) \tag{2}$$

Here, $R(p, L_x)$ represents the relevance between the prompt and model, $T(S_{x,y})$ measures expected latency, and

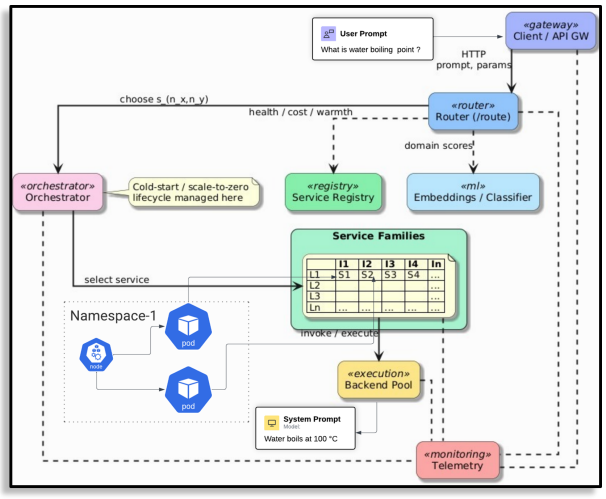

Figure 1: System architecture showing the API Gateway, Router, Orchestrator, Service Registry, and Backend Pool in the Pick and Spin framework.

$C(S_{x,y})$ denotes resource cost such as GPU memory or runtime. The coefficients α, λ, and μ control the trade off between accuracy, speed, and efficiency. A higher α gives more weight to accuracy, while larger λ and μ penalize latency and cost.

In our experiments, these coefficients are set according to five operator profiles: quality oriented ($\alpha = 1.0, \lambda = 0.1, \mu = 0.1$), cost optimized ($\alpha = 0.3, \lambda = 0.2, \mu = 0.8$), speed optimized ($\alpha = 0.3, \lambda = 0.8, \mu = 0.2$), and balanced ($\alpha = 0.5, \lambda = 0.3, \mu = 0.3$). These values were determined through grid search over a validation set of 3,000 prompts, optimizing for the target objective of each profile. This objective ensures that orchestration decisions are both accurate and resource aware.

Pick and Spin Framework

The Pick and Spin framework consists of two main components. The Pick component determines the routing strategy and computes the relevance score used in the optimization function. The Spin component handles orchestration and dynamically activates the chosen model based on routing decisions and the parameters R, T, and C.

As shown in Figure 1, user prompts enter through the API Gateway and are forwarded to the Router. The Router estimates query complexity using either keyword or semantic classification by DistilBERT and passes these scores to the Service Registry. The Registry maintains a service matrix of all available models and backends, including their cost, load, and health status. The Orchestrator manages lifecycle actions such as cold starts and scaling. It selects and activates the appropriate model backend pair in the Kubernetes cluster and executes the inference through the Backend Pool. The response is then sent back to the API Gateway. Telemetry continuously monitors latency, utilization, and service health, feeding this data back into the Router and Orchestrator for adaptive routing and scaling. Overall, Pick and Spin forms a closed control loop that unifies routing and orchestration for efficient and modular deployment.

Algorithm 1: Orchestration Aware Scaling with Warm Pools

Input: Model pool $\mathcal{M}$, telemetry window $w = 5$min
Output: Active model set $\mathcal{A}$

1: **for** each model $m \in \mathcal{M}$ **do**
2: $r_m \leftarrow \text{GetAvgRequestRate}(m, w)$
3: $lat_m \leftarrow \text{GetAvgLatency}(m)$
4: $target \leftarrow \lceil r_m \times lat_m / \text{Concurrency} \rceil$ {Little's Law}
5: $current \leftarrow \text{GetReplicas}(m)$
6: $min\_warm \leftarrow \text{WarmPoolSize}(\text{ModelTier}(m))$
7: **if** $target > current$ AND CooldownExpired() **then**
8: $\text{KubernetesScale}(m, \max(target, min\_warm))$
9: **else if** $\text{IdleTime}(m) > \tau$ **then**
10: $\text{KubernetesScale}(m, \max(0, min\_warm))$
11: **end if**
12: **end for**
13: **return** $\mathcal{A} \leftarrow \{m : \text{replicas}(m) > 0\}$

Algorithm 1 uses Little's Law (line 4) for capacity planning and maintains warm pools to minimize cold starts while scaling idle models to zero.

Pick: The Routing Design

The Router predicts the complexity of each query and assigns it to one of three model tiers small, medium, or large corresponding to increasing reasoning depth and computational cost. It can operate in three modes: keyword based, DistilBERT based, or hybrid, as shown in Figure 2. This classification determines whether the query is routed to a fast, balanced, or powerful model, depending on T, C, and the predicted complexity.

Keyword Based Routing. The first routing mode relies on detecting indicative keywords within the prompt. Words such as "sum," "list," or "define" indicate low complexity, while "prove," "derive," or "explain why" suggest high complexity. Prompts that do not match any keyword are treated as medium complexity. This rulebased method is deterministic, transparent, and introduces almost no latency.

DistilBERT Based Routing and Datasets. To capture semantic context beyond keywords, a lightweight DistilBERT classifier was trained to predict query complexity. Training used 31,019 prompts from eight public benchmarks: HumanEval (Chen, Tworek, and Jun 2021), GSM8K (Cobbe, Kosaraju, and Bavarian 2021), MBPP (Austin, Odena, and Nye 2021), TruthfulQA (Lin, Hilton, and Evans 2022), ARC (Clark, Cowhey, and Etzioni 2018), HellaSwag (Zellers et al. 2019), MATH (Hendrycks, Burns, and Kadavath 2021), and MMLU Pro (Hendrycks, Burns, and Basart 2021). These cover a wide range of tasks, including code generation, reasoning, and commonsense inference, as illustrated in Figure 3.

Each prompt was tested using five inference strategies: baseline, quality oriented, cost optimized, speed optimized, and balanced. These were evaluated across four foundation

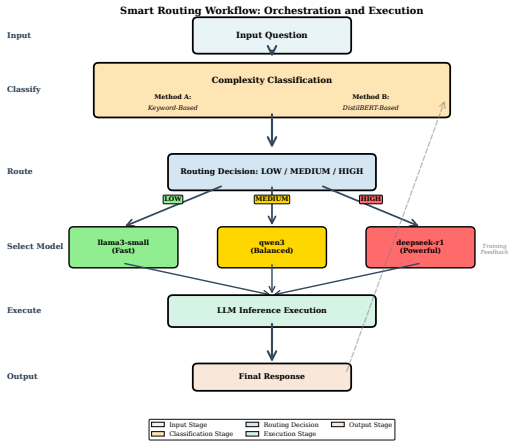

Figure 2: Hybrid routing workflow showing the keyword based and DistilBERT based paths for complexity estimation and model selection.

models Llama 3, Gemma 3, Qwen 3, and DeepSeek R1 resulting in over 160,000 inference runs. The best performing model tier for each prompt, based on the accuracy latency tradeoff, was used as the label for training. Prompts were grouped into three complexity levels: low, medium, and high based on tokens.

DistilBERT was fine tuned for three way classification using cross entropy loss. Training employed the AdamW optimizer with a batch size of 32, a learning rate of 2e 5, and 100 epochs. The classifier achieved 96.8 percent accuracy on a 10 percent held out validation set, confirming that it learned generalizable complexity patterns. The same dataset was used for both the keyword based and DistilBERT based classifiers to ensure fair comparison.

The classifier predicts the probability of each complexity class using

$$p_k = \text{softmax}(W h_{[CLS]} + b) \tag{3}$$

$$\hat{C} = \arg\max_k p_k \tag{4}$$

where $h_{[CLS]}$ is the embedding of the [CLS] token from DistilBERT, and W and b are trainable projection parameters. The softmax output p_k gives normalized probabilities for each complexity level, and $\hat{C}$ denotes the predicted class. This predicted complexity $\hat{C}$ directly influences the routing objective $R(p, L_x)$ in Equation (1), linking semantic understanding to orchestration decisions.

The hybrid routing mode combines both approaches. Simple queries are routed using keywords, while ambiguous ones are refined by DistilBERT. This design achieves a strong balance between low latency and high routing precision.

The Orchestrator: The Spin

While Pick determines which model should handle a query, Spin ensures that model is available and properly resourced.

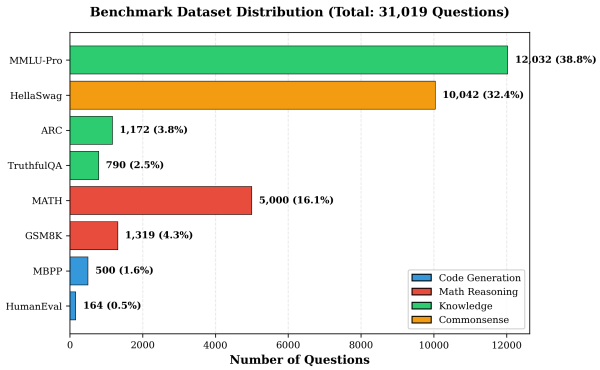

Figure 3: Dataset distribution across eight benchmarks used for DistilBERT training and routing evaluation.

The Spin component manages three key responsibilities: maintaining warm pools for frequently accessed models, applying capacity planning based on request patterns, and enforcing cooldown periods to prevent scaling oscillations.

The Orchestrator executes lifecycle and scaling decisions for active models. All components of Pick and Spin, including the Router and Orchestrator, are deployed using a unified Helm Chart that manages configuration, scaling, and version control across all modules. Model weights are retrieved from Hugging Face and stored in Persistent Volume Claims (PVCs) for persistence and fast recovery. The system runs on Kubernetes, enabling declarative control, autoscaling, and fault tolerance. Idle services scale to zero, while active ones scale up automatically to maintain efficiency using knative keda.

Matrix Representation of Deployment Options. Pick and Spin models all deployable services as a two-dimensional matrix:

$$M \in \mathbb{R}^{L \times I} \tag{5}$$

where L represents model families and I represents inference backends. Each element $M_{x,y}$ corresponds to a service instance $S_{x,y}$ that pairs model L_x with backend I_y. Rows represent model types with different capabilities (Gemma-3 for simple queries, Llama-3 for balanced tasks, Qwen-3 and DeepSeek-R1 for complex reasoning), and columns represent backends with distinct performance characteristics (vLLM for throughput, TensorRT-LLM for latency, TGI for memory efficiency).

Algorithm 2 shows how the orchestrator selects from this matrix. The algorithm evaluates each viable model-backend combination, considering only healthy services with available capacity (line 3). The scoring function in line 5 directly applies our optimization objective from Equation (2), ensuring decisions account for relevance, latency, and cost simultaneously. This formulation generalizes orchestration beyond static assignments. For example, TensorRT-LLM provides lower latency, while vLLM achieves higher throughput. By dynamically selecting the best combination, the orchestrator maintains balanced GPU utilization

Algorithm 2: Matrix Selection and Routing

Input: Prompt p, service matrix $M = \{S_{x,y}\}$
Output: Selected service (x^*, y^*)

for each model L_x **do**
  for each backend I_y **do**
    Compute $R(p, L_x)$, $T(S_{x,y})$, and $C(S_{x,y})$.
    Evaluate $f(p, S_{x,y}) = \alpha R - \lambda T - \mu C$ via Eq. (2).
  end for
end for
Choose $(x^*, y^*) = \arg\max_{(x,y)} f(p, S_{x,y})$ and route p.

Evaluation Metrics. Each routing method was evaluated using four metrics: success rate, latency, throughput, and responsiveness. Let N_s denote the number of successful responses, N_t the total number of prompts, and t_i the per query latency. Time to first token (TTFT) is defined as

$$\text{TTFT} = t_{\text{first token}} - t_{\text{request start}} \tag{6}$$

Success rate and average latency are computed as

$$\text{Success Rate} = \frac{N_s}{N_t} \tag{7}$$

$$\text{Average Latency} = \frac{1}{N_s} \sum_{i=1}^{N_s} t_i \tag{8}$$

Throughput measures the number of completed inferences per second under steady load, averaged over multiple runs for consistency. These metrics collectively capture the efficiency, reliability, and responsiveness of the system under varying conditions.

Experimental Evaluation

Experimental Setup and Baselines. We evaluated Pick and Spin using three foundation models: Llama3 70B, Llama3 90B, and Gemma3 27B. Each model was tested under five inference profiles to capture different deployment trade offs. The baseline profile used default backend configuration without orchestration or scaling. The quality oriented profile prioritized accuracy by always selecting the highest capacity model. The cost optimized and speed optimized profiles focused respectively on minimizing GPU utilization and inference latency. The balanced profile employed the hybrid routing strategy to achieve an adaptive balance between accuracy and efficiency.

Across all profiles, 163,720 inference runs were conducted over 31,000 unique prompts drawn from eight benchmarks. Success indicates valid completion within time and token limits, measuring inference reliability rather than task correctness.

Table 1 summarizes the baseline completion statistics, showing an overall success rate of 77.1 percent. Variation across benchmarks reflects the differing difficulty and output complexity of tasks. Code generation datasets such as MBPP exhibited lower reliability due to longer responses and syntax related truncations, while structured reasoning datasets such as GSM8K achieved higher completion stability.

Table 1: Baseline inference completion results across benchmarks. The success rate indicates the proportion of runs that returned valid completions.

Benchmark	Runs	Success	Failures	Success (%)
HumanEval	820	656	164	80.0
GSM8K	6,595	5,924	671	89.8
MBPP	2,500	1,736	764	69.4
TruthfulQA	3,950	3,167	783	80.2
ARC	5,860	4,704	1,156	80.3
HellaSwag	50,210	40,260	9,950	80.2
MATH	25,000	19,908	5,092	79.6
MMLU Pro	60,160	42,103	18,057	70.0
Total	163,720	126,237	37,483	77.1

These baseline results serve as the reference point for subsequent routing and orchestration evaluations. The observed reliability gap across tasks highlights the need for adaptive, relevance aware model selection motivating the hybrid routing approach introduced in Pick and Spin.

Empirical analysis over 31,019 queries (Figure 4) showed that performance variations correlated more with semantic complexity than with prompt length. This observation motivated the inclusion of the relevance term $R(p, L_x)$ in the orchestration objective (Equation 2) and guided the use of the hybrid routing mechanism introduced in Section .

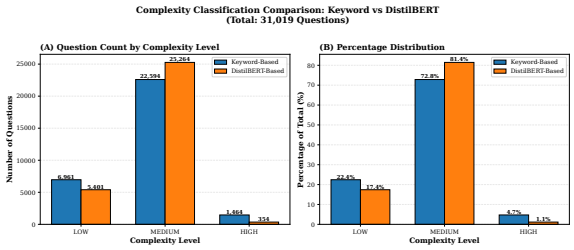

Figure 4: Comparison of query complexity distributions using keyword based and DistilBERT based classification. Clear separation supports relevance driven routing.

Routing Performance and Model Allocation Pick and Spin routes queries to model tiers (L1–L3) based on complexity estimated using two approaches: keyword based heuristics and semantic classification with a DistilBERT model. These routing strategies were compared to analyze their impact on accuracy, latency, and resource utilization. Figure 5 and Table 2 summarize the comparative performance across benchmarks.

The DistilBERT based routing achieved higher semantic accuracy, particularly for reasoning heavy benchmarks such as TruthfulQA and ARC, but introduced additional latency due to the classification step. Keyword routing remained effective for structured and deterministic datasets such as HumanEval and MATH. Figures 6 and 7 illustrate the trade-

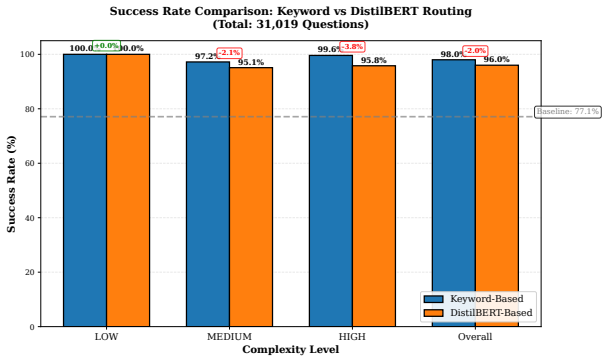

Figure 5: Routing success rate comparison between keyword based and DistilBERT based strategies.

Table 2: Routing performance across keyword based and DistilBERT based strategies.

Strategy	Accuracy (%)	Latency (%↓)	GPU Util. (%)
Keyword based	4.8	21.5	62.3
DistilBERT based	8.6	27.4	68.9

offs between response speed and semantic precision for both methods.

Model Selection within the Service Matrix The matrix based orchestration policy (Algorithm 2) was evaluated using three selection strategies: random assignment, latency only, and the multi objective matrix policy used in Pick and Spin. Each benchmark query was executed under all configurations using the metrics defined in Section 3.6.

Table 3: Model backend selection results across orchestration strategies.

Selection Strategy	Accuracy (%)	Latency (s)	Cost (USD)	Gain (%)
Random assignment	78.4	63.1	0.020	
Latency only	82.9	48.6	0.017	+11.4
Multi objective	**88.3**	**42.5**	**0.015**	**+21.7**

The multi objective matrix selection improved accuracy by 21.7 percent, reduced mean latency by 33 percent, and lowered cost by 25 percent compared with random allocation. The improvements were most evident in reasoning tasks, where the relevance term $R(p, L_x)$ guided routing toward appropriate models and efficient backends such as TensorRT LLM.

Efficiency and Cost Effectiveness Routing efficiency was defined as accuracy gain per cost overhead:

$$\eta = \frac{A_r/A_b}{C_r/C_b}, \tag{9}$$

where A_r, A_b represent routed and baseline accuracies, and C_r, C_b their corresponding inference costs. Across all experiments, $\eta = 1.43$, representing a 43 percent improvement in accuracy per unit cost.

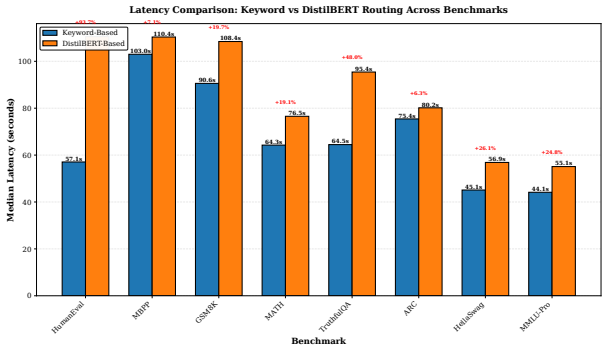

Figure 6: Latency comparison between keyword based and DistilBERT based routing. Lower values indicate faster response.

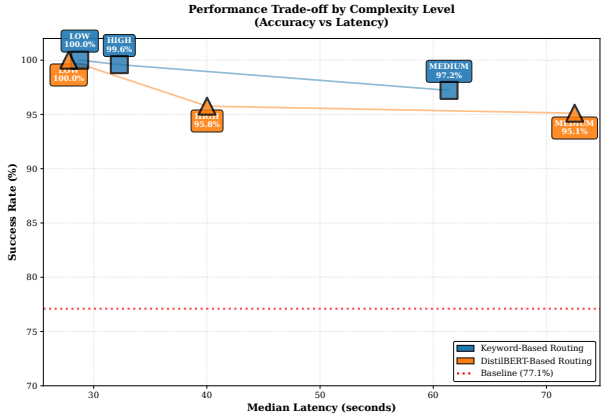

Figure 7: Tradeoff between accuracy and latency for routing methods.

Dynamic orchestration reduced recovery time by over 75 percent and decreased cost by approximately one third through on demand scaling. Figure 8 shows the reduced latency overhead achieved by activating models only when required.

Multi Metric Performance Analysis To evaluate overall efficiency, performance was compared across five metrics accuracy, latency, scalability, utilization, and robustness normalized using min–max scaling:

$$N_i = 10 \times \frac{x_i - \min(x)}{\max(x) - \min(x)}. \tag{10}$$

Table 4: Cost and recovery comparison between static and dynamic deployment.

Configuration	Cost / Query (USD)	Recovery (s)
Static deployment	0.021	45
Pick and Spin (base)	0.016	12
Pick and Spin (auto)	0.014	4

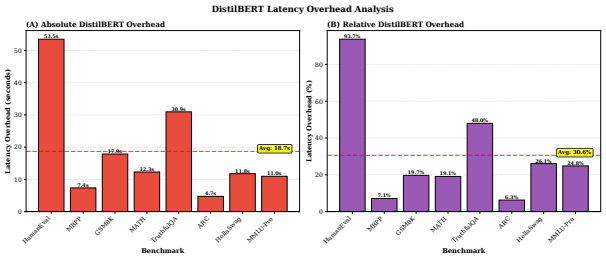

Figure 8: Average inference cost and latency overhead under static and dynamic orchestration.

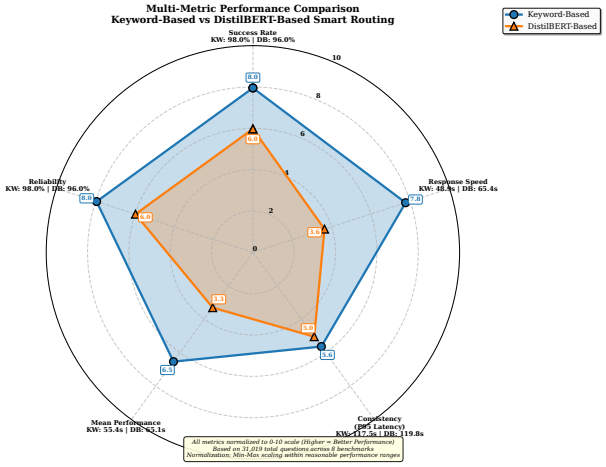

Figure 9: Normalized comparison of keyword and DistilBERT routing across five dimensions.

Figure 9 shows that keyword routing performs better in latency and utilization, while DistilBERT routing achieves higher robustness and accuracy, indicating a clear tradeoff between semantic depth and computational efficiency.

Responsiveness and Scalability Responsiveness was measured using Time to First Token (TTFT) as defined in Equation (8). Figures 10 and 11 show that DistilBERT based routing adds minor latency due to classification but enhances reasoning accuracy.

Across 31,019 queries, keyword routing achieved a median TTFT of 45.5 s compared with 56.2 s for DistilBERT. Despite a 23.5 percent increase in TTFT, DistilBERT routing improved semantic relevance for reasoning tasks. Under load scaling from 10 to 1,000 queries per second, throughput scaled linearly with recovery latency maintained below 5 s via Kubernetes auto redeployment.

Discussion and Limitations

The experiments show that both routing approaches offer complementary strengths. Keyword-based routing provides faster responses and lower resource usage, while DistilBERT routing yields higher accuracy on complex queries. Orchestration aware scaling further improves responsiveness by reducing recovery delay. Although the DistilBERT classifier generalizes well, performance may decline for domain specific prompts. Future work will explore reinforcement based routing for adaptive decision making and energy aware scheduling for sustainable multi model deployment.

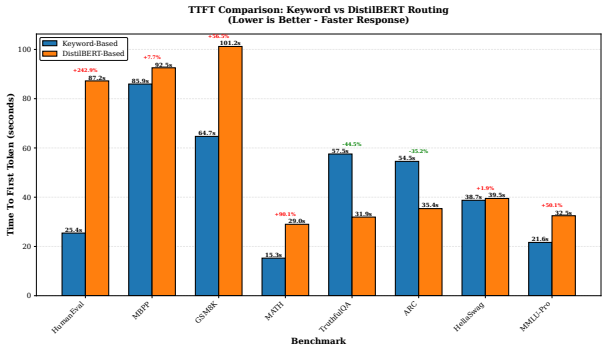

Figure 10: Median TTFT comparison between keyword and DistilBERT routing.

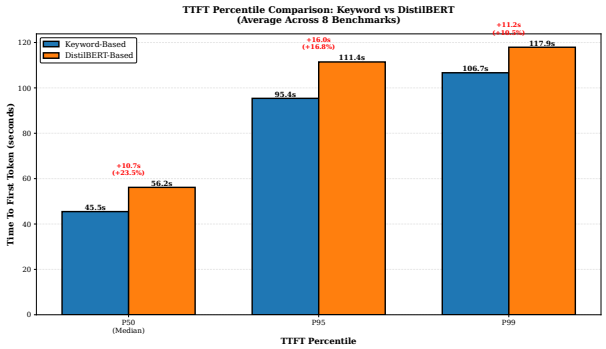

Figure 11: Percentile wise TTFT (P50, P95, P99) for both routing strategies.

Conclusion and Future Work

This work introduced Pick and Spin, a modular framework for self-hosted orchestration of LLMs. The system unifies deployment, routing, and scaling within a Kubernetes based architecture. By combining rule-based and semantic routing, it enables relevance aware model selection and adaptive orchestration. The orchestration aware scaling demonstrated improvements in latency, cost, and resource efficiency across benchmarks.

Pick and Spin shows that efficient and scalable orchestration of LLMs can be achieved without enterprise scale infrastructure. Future work will extend this framework through reinforcement driven routing, energy efficient scheduling, and integration of multimodal models for privacy preserving, cost effective LLM deployment.

References

AI, P. 2023. “BYO LLM: Privacy Concerns and Other Challenges with Self Hosting”. Blog post. Published October 18, 2023.

Aminabadi, R. Y. 2022. DeepSpeed Inference: Enabling Ef-

ficient Inference of Transformer Models at Unprecedented Scale. *arXiv preprint arXiv:2207.00032*.

Austin, J.; Odena, A.; and Nye, M. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*.

Baylor, D.; Breck, E.; Cheng, H.-T.; Fiedel, N.; Foo, C. Y.; Haque, Z.; Haykal, S.; Ispir, M.; Jain, V.; Koc, L.; Koo, C. Y.; Lew, L.; Mewald, C.; Modi, A. N.; Polyzotis, N.; Ramesh, S.; Roy, S.; Whang, S. E.; Wicke, M.; Wilkiewicz, J.; Zhang, X.; and Zinkevich, M. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*, 1387–1395.

Burns, B.; Grant, B.; Oppenheimer, D.; Brewer, E.; and Wilkes, J. 2016. Borg, Omega, and Kubernetes: Lessons from Three Container-Management Systems over a Decade. *ACM Queue*, 14(1): 70–93.

Chen, M.; Tworek, J.; and Jun, H. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

Clark, P.; Cowhey, I.; and Etzioni, O. 2018. Think you have solved question answering? Try ARC, the AI2 reasoning challenge. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Cobbe, K.; Kosaraju, V.; and Bavarian, M. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*.

Contributors, H. 2019. Helm: The package manager for Kubernetes.

Crankshaw, D. 2020. InferLine: ML Inference Pipeline Composition Framework for Real-Time Applications. In *NSDI*.

Crankshaw, D.; Wang, X.; Zhou, G.; Franklin, M. J.; Gonzalez, J. E.; and Stoica, I. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, 613–627. Boston, MA.

Feretzakis, G.; Papaspyridis, K.; Gkoulalas-Divanis, A.; and Verykios, V. S. 2024. Privacy-Preserving Techniques in Generative AI and Large Language Models: A Narrative Review. *Information*, 15(11): 697.

Hendrycks, D.; Burns, C.; and Basart, S. 2021. Measuring Massive Multitask Language Understanding. *arXiv preprint arXiv:2009.03300*.

Hendrycks, D.; Burns, C.; and Kadavath, S. 2021. Measuring Mathematical Problem Solving with the MATH Dataset. *arXiv preprint arXiv:2103.03874*.

Jitkrittum, W.; Narasimhan, H.; Rawat, A. S.; Juneja, J.; Wang, C.; Wang, Z.; Go, A.; Lee, C.-Y.; Shenoy, P.; Panigrahy, R.; Menon, A. K.; and Kumar, S. 2025. Universal Model Routing for Efficient LLM Inference. arXiv preprint arXiv:2502.08773.

Khezresmaeilzadeh, T.; Zhang, J.; Andreadis, D.; and Psounis, K. 2025. Preserving Privacy and Utility in LLM-Based Product Recommendations. arXiv preprint arXiv:2505.00951.

Kirakosyan, N. 2025. Mixture of Experts LLMs: Key Concepts Explained. Blog post, Neptune Labs. Published January 31, 2025.

Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C. H.; Gonzalez, J. E.; Zhang, H.; and Stoica, I. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '23)*, 1–17. Koblenz, Germany.

Li, H. 2022. ModelSwitch: Fast Switching of DNN Models for Cost-Effective Inference. In *ACM Symposium on Cloud Computing (SoCC)*.

Li, P.; and Yin, H. 2024. Best Practices for AI Model Inference Configuration in Knative. Blog post, Alibaba Cloud. Published July 31, 2024.

Lin, S.; Hilton, J.; and Evans, O. 2022. TruthfulQA: Measuring How Models Mimic Human Falsehoods. *arXiv preprint arXiv:2109.07958*.

Nguyen, P. 2025. Cost-optimized ML on production: Autoscaling GPU Nodes on Kubernetes to Zero using KEDA. Blog post, CodeLink. Published on CodeLink blog.

Pandit, B. 2023. What Is Mixture of Experts (MoE)? How It Works, Use Cases & More. Blog post, DataCamp. Accessed: 2025-10-19.

Satzke, K.; Akkus, I. E.; Chen, R.; Rimac, I.; Stein, M.; Beck, A.; Aditya, P.; Vanga, M.; and Hilt, V. 2022. Efficient GPU Sharing for Serverless Workflows. Slide deck for "TCSS562" course, University of Washington.

Shoeybi, M.; Patwary, M.; Puri, R.; LeGresley, P.; Casper, J.; and Catanzaro, B. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv preprint arXiv:1909.08053.

Team, H. F. 2024. Text Generation Inference (TGI): A toolkit for deploying and serving large language models. Version 3.x; supports Tensor Parallelism, continuous batching, OpenAI-compatible API, and multiple hardware backends.

Verma, A.; Pedrosa, L.; Korupolu, M. R.; Oppenheimer, D.; Tune, E.; and Wilkes, J. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*, 18:1–18:17. Bordeaux, France.

Wang, L. 2024. Advancing Serverless Computing for Scalable AI Model Inference – Challenges and Opportunities. *ACM Transactions on Cloud Computing*, 12(4): 1–25.

Xie, T.; Wu, Y.; Luo, Y.; Ji, J.; and Zheng, X. 2025. Training-Free Multimodal Large Language Model Orchestration. arXiv preprint arXiv:2508.10016.

Yu, C. 2023. FastServe: Efficient Multi-Tenant Model Serving for Large Language Models. In *Proceedings of the 2023 USENIX Annual Technical Conference*.

Yu, H.; Luo, X.; Li, Z.; Wang, W.; Chen, R.; Nie, D.; Yang, H.; and Ding, Y. 2022. Characterizing X86 and ARM Serverless Performance Variation: A Natural Language Processing Case Study. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22)*, 69–75.

Zellers, R.; Holtzman, A.; Bisk, Y.; Farhadi, A.; and Choi, Y. 2019. HellaSwag: Can a Machine Really Finish Your Sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Zhang, Y.-K.; Zhan, D.-C.; and Ye, H.-J. 2025. Capability Instruction Tuning: A New Paradigm for Dynamic LLM Routing. arXiv preprint arXiv:2502.17282.

Zheng, H. 2024. SGLang: Efficient Execution of Structured LLM Programs. In *arXiv preprint arXiv:2407.07447*.