arXiv:submit/7698361 [cs.AI] 11 Jun 2026

xMIx: High-Performance Serving-Time Platform for Mechanistic Interpretability Apps

Michael Blum
Technion

Mark Silberstein
Technion

Yaniv David
Technion

Abstract

Mechanistic interpretability (MI) has emerged as a powerful approach for analyzing and intervening in inference computations, with a growing number of applications such as jailbreak attempt detection, truthfulness evaluation, and hallucination detection. Unfortunately, MI deployment in production model-serving systems is currently not practical, as most existing MI frameworks introduce prohibitively high runtime overheads. The fundamental problem is that MI functions do not compose cleanly with served models: they fragment deployment, often force draining requests and rebuilding serving state, and conflict with critical performance optimizations such as continuous batching and CUDA-graph execution, essential for production deployments.

We present xMIx, a serving-native framework for deploying MI applications in production inference serving environments. xMIx enables *attaching* MI functions to a predefined set of locations in the model runtime, interposing on activations within the layers and residual streams. xMIx supports conditional invocation of MI functions depending on the outputs in preceding model layers. Multiple MI applications can be deployed in a single model instance. xMIx compiles them all into the serving path but activates them *dynamically at runtime* only when necessary, with negligible performance cost, and without requiring a separate model instance or alternative execution stack.

We integrate xMIx with the `vLLM` serving system and evaluate it across three major models and seven diverse MI applications. xMIx achieves performance comparable to native `vLLM` execution, incurring a slowdown of 1.3% mean inter-token latency (ITL), 1.2% for tail P99 ITL, 2.6% for mean time to first token (TTFT), and 1.6% for mean total token throughput (TTT).

1 Introduction

Mechanistic Interpretability (MI) has emerged as a promising approach for understanding and controlling the behavior of Large Language models (LLMs). Instead of treating models as opaque black boxes and reason only about inputs and outputs, MI operates directly on the model's internal computations during inference. By inspecting, tracing, and intervening on intermediate activations and residual streams, MI techniques aim to gain insight into how models produce specific behaviors and, increasingly, to modify those behaviors at runtime.

There is a growing class of *MI apps*, such as jailbreak detection [Kadali and Papalexakis, 2026], hallucination mitigation [Li et al., 2023], truthfulness evaluation [Orgad et al., 2025], safety monitoring [Lee et al., 2025], and controllable generation [Stickland et al., 2024]. More broadly, MI promises to become a practical mechanism for *augmenting already deployed models* with new runtime functionality without retraining or fine-tuning. Instead of modifying model weights, MI apps operate on transient inference computations, enabling systems to inject new behaviors, suppress undesirable outputs, enforce safety constraints, or dynamically adapt responses during execution.

Preprint.

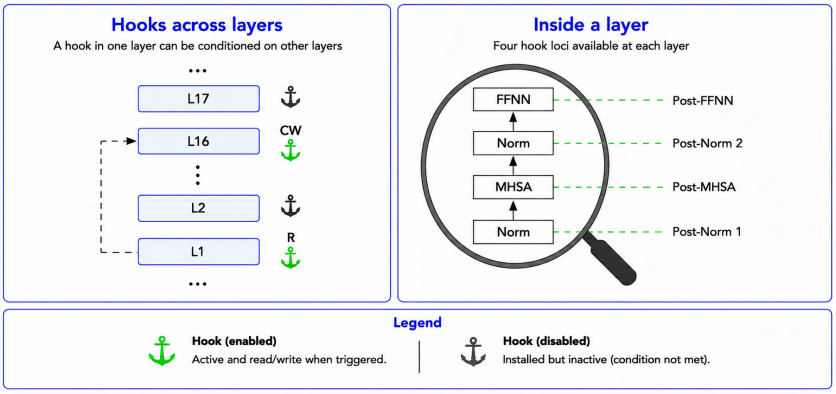


Figure 1: xMIx overview. MI functions can be installed (*attached*) in different transformer layers and in multiple loci (*hooks*) within a layer, and can read/write the respective activations and residual streams. At each locus, a function can be dynamically enabled/disabled, allowing easy implementation of common MI primitives, and seamless toggling of different MI functions in a single model deployment. In the figure, layer L16's activations are modified (conditional-write) based on the processing (read) of activations after L1.

Such runtime intervention mechanisms may become an important systems mechanism for extending model capabilities post deployment.

Unfortunately, most MI apps today remain confined to research prototypes and are rarely deployed in production serving stacks. A key reason is the *lack of high-performance runtime that meets strict efficiency targets of production deployments*. Existing MI frameworks (e.g., TransformerLens [Nanda and Bloom, 2022], and others discussed in Section 2), typically rely on intrusive instrumentation hooks and custom execution paths incompatible with modern inference systems. In production environments, where serving throughput and latency depend critically on optimizations such as continuous batching and CUDA graph execution, even lightweight MI logic that affects these optimizations can disrupt scheduling and significantly degrade performance. For example, recent vLLM-lens project reported about 20% throughput slowdown over vanilla `vLLM` for a simple steering logic applied to a single layer [vLL, 2026] (see Section 5 for more results).

Concurrently, the growing MI apps diversity creates another challenge: each app is often implemented as a separate inference pipeline modification, tightly coupled to a specific runtime, model and intervention mechanism. As a result, operators cannot efficiently deploy multiple MI apps simultaneously on the same served model. Instead, deploying new apps requires maintaining separate model instances, alternative execution stacks, or specialized serving infrastructure, rather than a unified control-plane for managing multiple interventions. This makes deployment much more complex: one needs to choose which MI apps are enabled in advance. Toggling of apps at runtime is prohibitively expensive: it forces the serving stack to be restarted, draining requests and rebuilding serving state.

The missing link. We argue that the missing link between MI research and production deployment is a serving-native abstraction layer that enables MI apps to compose cleanly with optimized inference runtimes. Such a layer is critical for making MI *actionable* [Orgad et al., 2026], enabling broader adoption and practical impact in production model serving systems.

We present xMIx, a lightweight serving-native framework for flexible and efficient deployment of MI apps in production inference environments. Figure 1 illustrates the main concepts. A developer specifies the *hooks* in the model where MI functions (GPU kernels in Triton) can be *attached* to implement the respective MI logic. Each function may access activations at the hook locus, modify them to influence the subsequent inference layers or/and save its results in a buffer that can be later accessed from the CPU or GPU code. In addition, MI function invocation can be *conditioned* on a predicate evaluated on the outputs from MI functions executed in earlier layers. For convenience, xMIx provides MI function templates to implement typical MI primitives – *read*, *write*, and *conditional-write*. Together, this simple, model-agnostic interface allows implementing a broad range of known MI apps, including 7 we demonstrate in our evaluation.

xMIx enables multiple MI apps on the same deployed model, rather than requiring separate execution stacks or dedicated model replicas. It compiles all the attached functions directly into the serving path, but activates them *only when required*. The governing principle is simple: if a deployed MI app is not active on a request, it should not impose meaningful performance cost on that request. Thus, a MI app can be disabled or enabled with negligible overheads, without disrupting the serving of pending inference requests.

Under the hood, xMIx employs several advanced techniques to meet its performance and flexibility goals. First, it interposes on CUDA graphs created by the serving frameworks to enable MI function attachment and toggling. Second, it adds token-triggered MI functions as well as data sharing across functions attached to different layers, while keeping negligible (a few MBs) memory state. Third, it supports divergent processing in a batch of tokens at each hook, without falling back to the CPU-driven execution. Last, it seamlessly supports multi-GPU execution, without any involvement of MI apps developers.

Results. We integrate xMIx with `vLLM` serving runtime, and implement seven MI apps with diverse MI primitives for three popular mid-size LLMs. Across all the configurations, xMIx demonstrates minor overheads for all key performance metrics, slowing down `vLLM` by 1.3% for mean ITL, 1.2% for tail P99 ITL, 2.6% for mean TTFT and 1.6% for mean TTT. These results not only show that xMIx overheads are at least an order of magnitude lower than in existing MI frameworks, but also that MI apps can be readily deployed in production serving systems with acceptable costs.

Contributions. Our key contributions are: (i) We analyze different MI applications and use cases, and generalize them, creating unified abstractions that allow MI developers to interact with the model under a general framework. (ii) We built a framework on top of `vLLM`, allowing the deployment of a broad spectrum of different apps, while preserving all the benefits of high-throughput, low-latency inference. (iii) We implement a dynamic toggling mechanism, which allows don't-use-don't-pay deployment, enabling or disabling applications as they are needed at runtime.

2 Motivation and Related work

MI techniques analyze and intervene on the computations performed inside a model, rather than only its inputs and outputs (see Table 4 for the list of surveyed apps). For autoregressive transformers, this usually means reading hidden states, attention values, or MLP activations at chosen layers and token positions, then either reporting a signal (e.g., a probe score) or writing back a modified activation. This pattern underlies a broad set of applications, including probing, activation patching, activation steering, and conditional interventions.

MI applications are not merely post-hoc analyses. They are inference-time programs that must execute in lockstep with the model's forward pass. Many MI interventions read an intermediate value at a specific layer and token position, then use that value to modify a later intermediate state in the same execution. For example, [Lee et al., 2025] reads activations at one layer and uses them to induce refusal at a later layer by writing a modified activation.

Unfortunately, despite their immense potential, broader deployment of MI applications in production model serving systems is limited due to high performance costs. We analyze the key reasons next.

Research tooling for MI. Most existing MI tooling was built for rapid research iteration, not for production inference. TransformerLens [Nanda and Bloom, 2022] popularized a convenient hook-based interface and standardized naming for transformer internals, making it easy to cache, edit, and replace activations. However, it does so as a PyTorch-level instrumentation library rather than as an integration with a serving engine [Paszke et al., 2019]. It uses PyTorch's *eager execution mode*, prioritizing flexibility over efficiency. Specifically, it allows arbitrary invocation of MI functions during inference, which is not compatible with CUDA graphs, and thus sacrifices performance due to the resulting frequent CPU-GPU synchronization. NNsight [Fiotto-Kaufman et al., 2024] generalizes this style with a flexible tracing API over PyTorch models and a remote backend, again prioritizing expressive experimentation over performance. EasyEdit [Wang et al., 2024] organizes a wide range of knowledge-editing methods behind a common framework, but its focus is offline or session-level model editing rather than low-latency, multi-tenant serving. EasyEdit2 [Xu et al., 2025c], EasySteer [Xu et al., 2025b], and more recently vLLM-lens [vLL, 2026] move control closer to `vLLM`'s serving engine by unifying steering-vector generation and application, yet they still treat steering as a model-level framework layered above inference and resort to PyTorch's eager execution

model. Thus, by targeting MI research and prototyping, these tools are incompatible with the strict performance goals of production serving systems.

Why production model serving is different. Modern LLM serving systems, such as `vLLM` maximize throughput by applying sophisticated techniques, including disaggregated execution of prefill and decode, continuous batching, explicit KV cache management, and GPU-oriented optimizations such as FlashAttention [Dao et al., 2022] and PagedAttention [Kwon et al., 2023]. They often rely on *CUDA Graphs* [NVIDIA, 2026a] to reduce recurrent kernel launch overheads, and introduce custom GPU kernels, including Triton-based fused kernels in `vLLM`, to keep performance-critical operations on the device fast path [Tillet et al., 2019, vLLM Project, 2026b]. These systems are designed to minimize host-device synchronization and to preserve fixed, CUDA-graph friendly execution paths.

Under these conditions, regular Python hooks are not an appropriate implementation approach. They force the serving stack back into a host-driven eager execution mode. Each intervention that escapes the GPU fast path, adds CPU participation on the critical path, introduces synchronization points, and often requires extra data movements. That cost directly conflicts with continuous batching and CUDA graphs: once execution must wait for Python, the system loses the fixed path that these optimizations rely on. The penalty is especially severe during decode, where serving is already memory-bound and even modest per-token overhead accumulates across long generations.

Seen from a systems perspective, existing MI frameworks make the same core assumption: they operate through CPU-driven execution, where tensor operations are dispatched immediately by the host program rather than staged into a serving-native execution graph [Paszke et al., 2019], with hooks, traces, or user-space callbacks layered around the model rather than integrated into the serving engine. TransformerLens [Nanda and Bloom, 2022], NNsight [Fiotto-Kaufman et al., 2024], EasyEdit [Wang et al., 2024], EasyEdit2 [Xu et al., 2025c], EasySteer [Xu et al., 2025b] and vLLM-lens [vLL, 2026] differ in interface and scope, but they all expose MI by interposing software around the forward pass Python flow instead of compiling interventions into the execution path itself. As we show later in Table 3, the overheads induced by this execution model render it unusable for high-throughput serving.

The missing MI runtime layer. The central gap preventing production deployment of MI is the lack of a production-ready programming and runtime model for MI. A deployable system must support many interventions inside the forward pass while preserving the execution constraints of the serving engine. Another crucial requirement for such systems is the ability to enable or disable an MI application without swapping a model, via mid-flight reconfiguration. Model replacement is too disruptive for performance and error-prone: operators must drain in-flight requests, preserve correctness for existing KV caches and captured execution graphs, warm up the replacement instance, and absorb temporary capacity loss and tail-latency spikes. These issues require system-level, rather than application-level runtime tightly integrated with the serving environment.

This work targets this missing layer: a general MI framework for production model serving systems with dynamic, low-overhead control over interventions.

3 Survey of representative MI applications

To scope the interface that xMIx should expose, we surveyed 14 recent MI applications (see Appendix A for the full list), and clustered them by the intervention patterns they require. From each cluster we selected a representative application to include in Table 1. The columns capture the

#	Paper	Purpose	Primitive	Trigger	Target
1	Single Direction (Arditi et al. [2024])	Safety	Write	Uncond.	All Layers
2	SEAL (Chen et al. [2025])	Reasoning	Cond. Write	Token	All Layers
3	Conditional Refusal (Lee et al. [2025])	Safety	Cond. Write	Activations	Multi-Layer
4	Hallucination Probe (Orgad et al. [2025])	Truthfulness	Read	Activations	Single-Layer
5	SAKE (Scialanga et al. [2025])	Knowledge	Write	Uncond.	Single-Layer
6	Side-Effect-Free Steering (Stickland et al. [2024])	Safety	Write	Uncond.	Multi-Layer
7	SteerMoE (Fayyaz et al. [2026])	Safety	Cond. Write	Router-Logits	Router-Logits

Table 1: Representative MI applications with primitives, triggers and intervention loci. Additional applications are surveyed in Table 4.

```
truth_probe = (m.read("probe")
  .layers([23])
  .submodule("residual.post")
  .cond("last_token"))
```

(a) Read-only intervention

```
guarded_steer = (
  m.write("steering_vector")
  .layers([22])
  .submodule("attention.post")
  .cond(
    when="token_in(trigger_list)",
    gate="probe(layer=18) >= 0"))
```

(b) Conditional write

Figure 2: Example MI functions using xMIx. The strings represent either the standard submodule names (e.g., `residual.post`), or external Triton kernels to invoke at the hook (e.g., `steering_vector`).

dimensions that matter for a serving interface: the *Purpose* of the method, the *Primitive* it requires, the *Trigger* that determines whether the method activates, and the *Target* locus where it reads or intervenes. This organization lets us compare otherwise different methods by the runtime capabilities they demand, rather than by their end-task alone.

The main conclusion is that these applications collapse to three primitives. **Read** covers methods that only consume activations. A representative example is a linear-regression probe that assigns a truthfulness score from last-token activations. **Write** covers unconditional steering methods that modify activation targets to influence behavior, e.g., add or subtract a vector. **Conditional Write** covers methods that perform some computation before deciding whether to intervene, where the condition can depend either on activations at another layer, or on token-level information before the forward pass begins.

Once applications are grouped under these primitives, method-specific implementation details can be abstracted behind a common interface, which in turn opens the door to systems optimizations. In particular, the table suggests that most representative applications do not require arbitrary Python hooks or unrestricted user code inside the forward pass. Instead, xMIx mainly needs efficient support for reading named activation slices, writing to a small set of intervention loci, and guarding those writes with lightweight predicates. This also clarifies what must have dynamic on-off toggling control on a live model instance: typically small runtime artifacts such as probes, vectors, thresholds, feature dictionaries, or router masks, rather than a new copy of the model.

4 Design and Implementation

4.1 Programming Interface

xMIx exposes a lightweight and clean API that allows easy implementation of a variety of MI apps, while specifically facilitating primitive classes identified in Section 3. The interface is designed to make the logical structure of a MI app explicit: what value is read or written, where in the model it applies, and under what condition it should be active. Rather than exposing arbitrary callbacks, xMIx asks the user to describe interventions in this restricted form so that the runtime can map them onto serving-native execution paths. Fig. 2 includes two illustrative examples.

Core operations. The interface centers on two constructors, `m.read(...)` and `m.write(...)`, which cover the surveyed primitive space. A read operation declares that xMIx should materialize a derived signal from an activation locus, such as raw activations or a probe output. A write operation declares that xMIx should inject or transform a value at an activation locus, such as a steering vector or another intervention payload. Conditional write is expressed as an ordinary write augmented with an explicit activation predicate.

Target specification. Each operation is refined by selectors identifying where it applies. `.layers(...)` specifies either a layers concrete set or a broader scope such as `all`, while `.submodule(...)` identifies the target computation within the layer (e.g., pre-attention or post-attention representation). This decomposition keeps the API close to how practitioners already describe interventions, while exposing the targeting information that the runtime needs in order to place the corresponding serving-side hooks.

Activation conditions. `.cond(...)` controls when an operation should run. The condition may be token-driven, e.g., activated when the current token belongs to a trigger set, or activation-driven, e.g., gated by a probe score computed at another layer. This is the mechanism that lets XMIX express both simple read and write operations and more structured conditional behaviors within the same programming model. Just as important, it makes the control structure explicit enough for the runtime to map these conditions to CUDA graph-internal parameter updates or conditional execution rather than to CPU-side orchestration.

4.2 Translation to Runtime Semantics

Translation objective. The programming interface is not executed as a collection of arbitrary callbacks around the forward pass. Instead, XMIX lowers each declaration into a small set of serving-native runtime artifacts whose behavior is known in advance. This is the key design choice that reconciles flexibility with production inference constraints. The user describes *what* signal to read or write, and XMIX determines *how* to realize that request using the runtime's existing execution path.

Hook mapping. The selectors `.layers(...)` and `.submodule(...)` resolve to a predefined hook surface in the model. A hook denotes a legal attachment point where XMIX may expose an activation slice to an MI function without requiring an arbitrary control transfer out of the serving engine. These hooks are the translation targets of the API. They are exposed at component boundaries in the execution graph, where activations pass from one stage of computation to the next, rather than inside the internal implementation of a component. This narrow hook surface is intentional. It defines the places where intervention is possible while avoiding arbitrary instrumentation inside performance-critical fused internals, which is what makes the interface compatible with optimized serving backends.

Executable units. Once an operation has been mapped to a hook, XMIX realizes it as an attached MI function, typically a Triton kernel. XMIX provides a library of pre-implemented kernel templates for common MI primitives, but users may also import a custom kernel when the built-in library does not cover their application. At a design level, the contract is simple. A function consumes the data slice (e.g., activations) at the hook, together with the runtime parameters associated with the app, and produces either a derived signal, an activation update, or both. Read-like operations therefore materialize data for later use, while write-like operations produce a payload that is applied at the target hook. In other words, the kernel operates on the activation view presented at the hook, not by instrumenting the internals of the upstream component implementation.

State and dataflow. XMIX uses explicit shared state to connect functions attached to different hooks. Read outputs are written into buffers that can be consumed later by another MI function, returned to CPU-side code, or reused by GPU-side logic in subsequent layers. This buffer-mediated dataflow is what lets XMIX express cross-layer applications such as conditional steering based on an earlier probe result. Additionally, it constrains composition to a small and analyzable interface: functions communicate through named runtime state, not through arbitrary inter-kernel control flow.

Activation and control semantics. The runtime distinguishes between two main condition classes. Token-driven conditions depend on properties of the current token stream, such as whether the current token belongs to a trigger set. Activation-driven conditions depend on previously materialized MI function outputs, such as a probe score or feature activation computed at another hook. The API deliberately exposes only conditions that can be lowered to runtime control without returning to CPU-side orchestration. This restriction is the price of portability across optimized serving paths, but it is also what makes the resulting control logic compatible with production execution.

Toggling semantics. Deployment and activation are separate concepts in XMIX. An application may be configured into the serving path, with its hooks, functions, and state artifacts registered in advance, yet remain inactive on a particular request. Toggling therefore changes whether the relevant runtime artifacts participate in the current execution, while leaving the surrounding serving path unchanged. This is the design basis of the "don't use, don't pay" principle: if an application is inactive for a request, it should impose at most negligible overhead on that request.

Composition model. The same translation scheme allows multiple MI apps to coexist on one model instance. Each app contributes hook-attached functions together with the state and control artifacts they require, while XMIX provides a common runtime substrate that determines when each artifact is active. In that sense, XMIX acts as a control plane for MI apps rather than as a single-purpose intervention library. Different apps can therefore share one optimized deployment instead of requiring separate model replicas or ad hoc execution stacks.

4.3 Implementation

We implement xMIx on top of vLLM and use its serving abstractions to realize the design above through GPU-resident execution paths that remain compatible with the engine's optimization strategy. At this stage, the questions are no longer what hooks, functions, or buffers mean in the abstract, but how those abstractions are realized inside a concrete serving stack while preserving its performance properties.

vLLM integration surface. xMIx interposes on the execution path that vLLM uses to run model layers and inserts intervention points through the same generalized model abstractions that already support multiple model families. This integration surface is the concrete realization of the abstract hook model described above. It keeps the user-facing interface uniform, while allowing the backend to resolve hook placement through serving-side code rather than through a model-specific user API.

CUDA-graph-safe kernel design. To keep intervention overhead low, xMIx realizes selected primitive operations through hand-crafted GPU kernels, including Triton kernels for performance-critical cases. This lets xMIx fuse sequences of operations that frequently appear together in MI apps, reducing kernel-launch overhead and avoiding unnecessary intermediate materialization. The same design also serves a second purpose: CUDA-graph capture and replay require a stable execution structure with known operations and tensor layouts ahead of time. xMIx therefore implements its intervention paths so that the common cases remain CUDA-graph safe rather than forcing the serving engine to fall back to a less optimized execution mode. In particular, xMIx separates static graph structure from per-request dynamic parameters. When the topology of an intervention path is fixed but the kernel arguments vary, xMIx can reuse the same instantiated graph and update only the relevant kernel-node parameters before launch, instead of recapturing or rebuilding the whole graph. This follows the CUDA-graph pattern in which explicit node handles are retained for the dynamic kernels while the surrounding graph is captured once and replayed many times [NVIDIA, 2026a,b].

Graph-level toggling and continuous batching. On top of these kernels, xMIx integrates intervention logic into the execution graph used by the serving runtime. This allows the system to treat MI operations as first-class nodes in the serving path rather than as external callbacks. The important point is that not all dynamism is the same. Some MI applications require only dynamic parameters on a fixed execution path, while others require true data-dependent control flow. For the former, xMIx reuses an instantiated graph and updates the parameter state of the affected nodes at low cost before launch. For the latter, xMIx structures the graph so that an upstream GPU computation determines whether a downstream intervention body should execute, rather than returning control to the CPU to make that decision. This is conceptually aligned with CUDA conditional nodes, where a predicate computed on the device governs the execution of a nested graph region [Gaiser et al., 2024, NVIDIA, 2026b].

This distinction matters when for efficient support of continuous batching, where different requests may activate different interventions, leading to divergent compute paths during the graph execution. xMIx therefore aims to express request-selective behavior as graph-internal control, so that inactive applications are bypassed and active ones trigger only their relevant subgraph. The goal is to preserve a single serving-native execution path for the batch, without forcing every request to pay for the most expensive intervention branch or fragmenting execution into CPU-orchestrated special cases.

4.4 Current Scope and Limitations

The implementation is designed to be model-agnostic at the interface level, but the current realization is still tied to the model families and serving paths that we have integrated in vLLM. The general hook substrate is meant to extend across architectures while preserving the same programming model, yet this should not be read as a claim that every model is already wired up automatically or that every hook is available on every backend. Likewise, multi-GPU execution is handled by the serving runtime rather than exposed to the MI developer, but the portability claim here is practical rather than absolute: it depends on the coverage of the underlying vLLM integration.

Similarly, the current system supports intervention only at the exposed hook surface described above. Those hooks live at component boundaries in the serving path, not inside a component's internal implementation or a fused kernel. As a result, xMIx does not support intra-component or intra-kernel manipulations. This is a consequence of the same serving-native execution model that enables graph-safe integration, rather than a missing engineering feature. More generally, xMIx supports

only MI logic expressible through the exposed hooks, shared runtime state, and graph-safe kernels. Although XMIX provides a library of common kernels, unsupported finer-grained logic still requires a different integration strategy and lies outside the system's current scope. In particular, this limitation prevented us from implementing the jailbreak prevention mechanism of [Ben-Tov et al., 2025].

5 Evaluation

To assess XMIX's practicality, we deploy seven representative MI applications under a production-like benchmarking scenario and quantify the performance impact of running MI apps.

Setup. We evaluate on `Apps #1-#7`, corresponding to the indices in Table 1. `Apps #1-#6` are run on Llama-3.1-8B-Instruct, Mixtral-8x7B-Instruct-v0.1, and Qwen3-8B; the MoE-specific `App #7` is run on Mixtral only. We benchmark serving performance using `vLLM`'s `bench sweep` utility [vLLM Project, 2026a].

Workload. Each run issues 250 prompts under Poisson arrival, sampled from ShareGPT V3 unfiltered [ShareGPT, 2026]. For models without precomputed steering vectors, we introduce a random steering vector of matching dimension. To control the affect of steering, We fix output length at 128 tokens per prompt, so the actual content of the steering vector doesn't impact the measurement.

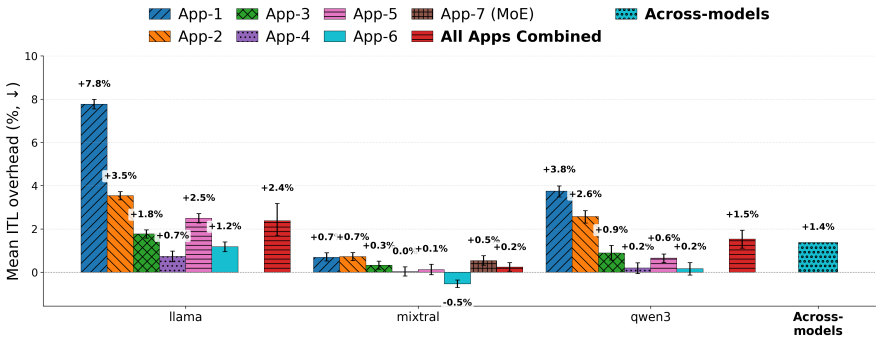
Baseline. Our measurement baseline is `vLLM` with the XMIX MI hooks *uninstalled*. Relative to vanilla `vLLM 0.11.1`, this configuration showed no measurable performance difference for all but one model. For that model, our version ran marginally *faster* while producing identical output; we traced this to memory-layout differences that yield a minor acceleration over the vanilla version. To ensure a fair comparison, we therefore report all results relative to the `vLLM` version with full XMIX support, rather than vanilla `vLLM`.

Hardware. We use a single AMD EPYC 7742 64-Core CPU server with NVIDIA NVLINK connected A100-SXM4-80GB GPUs. We run `vLLM`'s Tensor-Parallel configuration on two GPUs.

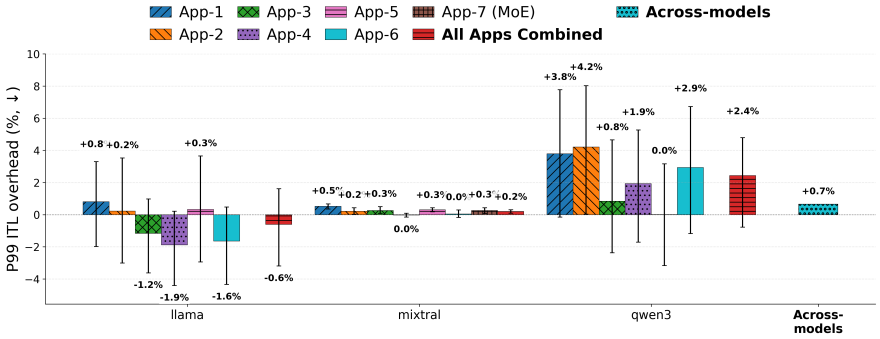
Metrics. We evaluate across the main performance metrics: ITL, tail ITL (P99), TTFT, and TTT. They provide a comprehensive view of the XMIX performance impact. TTFT indicates the added delay before generation begins; ITL and tail ITL indicate the added delay during decode; and TTT reflects the influence on the aggregate serving capacity.

Methodology. For each model, application, and metric, we discard the first three benchmark runs as warmup and aggregate the remaining 7 repeated runs into a single summary statistic for reporting. We report arithmetic mean for latency metrics such as mean ITL and p99 ITL, and harmonic mean for throughput. We then normalize by the baseline, and estimate 95% confidence intervals. We validate the output correctness of the platform when no steering is applied, via exact token comparison to the vanilla version. Where we had the correct steering vector (Llama-3.1-8b-Instruct with the refusal vector from [Arditi et al., 2024]), we also verified the steering works.

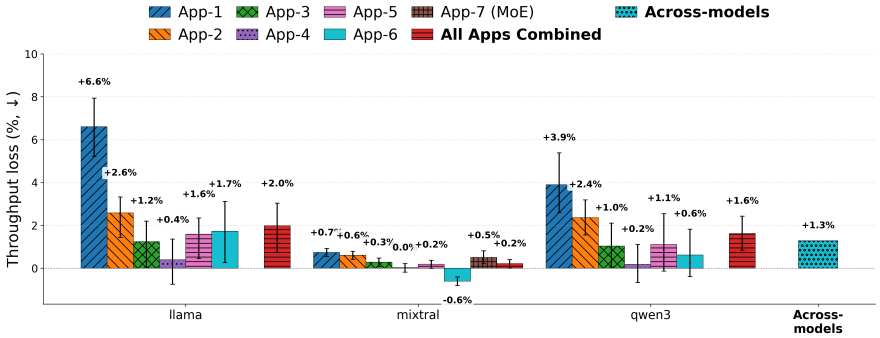
Results. Figure 3 summarizes the overheads introduced by MI apps on top of XMIX. Across all benchmarks, MI apps introduce only *minor performance degradation* with natural variation across applications and models. The overhead distribution across different MI apps is correlated with the amount of added processing incurred by each app. For example, `App-1` manipulates all model layers, and thus incurs the highest overhead across all models. In contrast, `App-4` intervenes in fewer layers for all tokens, and `App-3` is conditionally triggered for a subset of tokens, which in turn reduces its relative cost. Moreover, larger models with more layers (e.g. Mixtral-8x7B-Instruct-v0.1) and more parameters incur much lower relative overheads across apps due to amortized costs. We also notice a few peculiar phenomena, like `App-7`'s minor negative values (acceleration) on Mixtral. After deeper examination, we saw the performance dropped back down after removing unused arguments passed between steering kernels. So we attribute them to memory layout differences. We also noticed high variability of the P99 ITL metric (expressed by high error bars). Looking into it, we found that these runs usually showed slightly better throughput than the average run. We suspect that due to timing differences, more requests find themselves among mixed prefill-decode batches, raising ITL of the decode tokens in such passes compared to decode only batches, and driving P99 ITL of the whole run up.



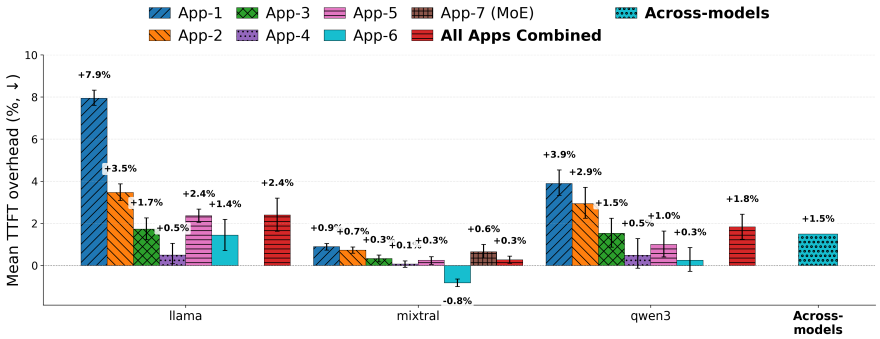
(a) Mean ITL



(b) Mean p99 ITL



(c) Mean TTT



(d) Mean TTFT

Figure 3: Overheads relative to the vLLM baseline for App#1-#7 deployed with xMIx. Each panel reports a per-model breakdown for Llama, Mixtral, and Qwen3, while the final bar summarizes the average change across models. Error bars denote confidence intervals.

Application	xMIx				EasySteer			
	Mean ITL (ms, ↓)	P99 ITL (ms, ↓)	Throughput (tok/s, ↑)	Mean TTFT (ms, ↓)	Mean ITL (ms, ↓)	P99 ITL (ms, ↓)	Throughput (tok/s, ↑)	Mean TTFT (ms, ↓)
App-1 (Refusal)	11.08 (+8%)	34.59 (+1%)	4639 (−7%)	7800 (+8%)	54.75 (+510%)	65.97 (+93%)	1488 (−82%)	25763 (+415%)
App-2 (SEAL)	10.65 (+4%)	34.39 (+0%)	4839 (−3%)	7476 (+3%)	25.71 (+186%)	30.74 (−10%)	3055 (−62%)	12798 (+156%)
App-3 (CAST)	10.46 (+2%)	33.91 (−1%)	4906 (−1%)	7351 (+2%)	26.00 (+190%)	29.16 (−15%)	3055 (−62%)	12701 (+154%)
App-5 (SAKE)	10.54 (+2%)	34.42 (+0%)	4888 (−2%)	7396 (+2%)	22.22 (+148%)	25.46 (−25%)	3533 (−56%)	11015 (+120%)

(a) Llama-3.1-8B-Instruct

Application	xMIx				EasySteer			
	Mean ITL (ms, ↓)	P99 ITL (ms, ↓)	Throughput (tok/s, ↑)	Mean TTFT (ms, ↓)	Mean ITL (ms, ↓)	P99 ITL (ms, ↓)	Throughput (tok/s, ↑)	Mean TTFT (ms, ↓)
App-1 (Refusal)	34.90 (+0.7%)	100.85 (+0.5%)	2550 (−0.7%)	16470 (+0.9%)	86.60 (+178%)	93.69 (+137%)	1009 (−62%)	40989 (+155%)
App-2 (SEAL)	34.91 (+0.7%)	100.54 (+0.2%)	2554 (−0.6%)	16443 (+0.7%)	41.31 (+33%)	47.77 (+21%)	2025 (−23%)	20767 (+29%)
App-3 (CAST)	34.77 (+0.3%)	100.59 (+0.3%)	2562 (−0.3%)	16378 (+0.3%)	43.48 (+40%)	49.30 (+25%)	1918 (−27%)	21985 (+37%)
App-5 (SAKE)	34.70 (+0.1%)	100.63 (+0.3%)	2564 (−0.2%)	16366 (+0.3%)	39.90 (+28%)	44.84 (+13%)	2078 (−21%)	20296 (+26%)

(b) Mixtral-8x7b

Application	xMIx				EasySteer			
	Mean ITL (ms, ↓)	P99 ITL (ms, ↓)	Throughput (tok/s, ↑)	Mean TTFT (ms, ↓)	Mean ITL (ms, ↓)	P99 ITL (ms, ↓)	Throughput (tok/s, ↑)	Mean TTFT (ms, ↓)
App-1 (Refusal)	10.98 (+3.8%)	36.13 (+3.8%)	4728 (−3.9%)	7983 (+3.9%)	63.33 (+554%)	86.10 (+161%)	1319 (−83%)	29656 (+439%)
App-2 (SEAL)	10.85 (+2.6%)	36.28 (+4.2%)	4804 (−2.4%)	7909 (+2.9%)	30.52 (+215%)	37.09 (+13%)	2649 (−66%)	15062 (+174%)
App-3 (CAST)	10.68 (+0.9%)	35.11 (+0.8%)	4869 (−1.0%)	7801 (+1.5%)	32.19 (+232%)	36.63 (+11%)	2538 (−67%)	15448 (+181%)
App-5 (SAKE)	10.65 (+0.7%)	34.82 (+0.0%)	4866 (−1.1%)	7761 (+1.0%)	27.54 (+184%)	31.46 (−4%)	2956 (−62%)	13387 (+143%)

(c) Qwen3-8B

Table 2: Performance of EasySteer and xMIx across four applications evaluated on multiple models, with signed percentage change against the vLLM baseline shown in parentheses. Latency metrics (Mean ITL, P99 ITL, Mean TTFT) are reported in ms (↓); throughput is in tokens/sec (↑).

Comparing to EasySteer. Setup. We conduct a direct comparison against EasySteer [Xu et al., 2025a] on the applications implemented on both platforms, App-1 (Arditi et al. 2024]), App-2 (Chen et al.[2025]), App-3 (Lee et al. [2025]), App-5 (Scialanga et al. [2025]), across the same models benchmarked in Figure 3, comparing to vLLM vanilla baseline. We adapted the EasySteer applications to work with the standard vLLM bench utility. In order to align with EasySteer app configuration, the reference vLLM vanilla baseline for EasySteer is run with prefix caching disabled and chunked prefill disabled and on a newer vLLM version. xMIx comparison is against vLLM vanilla baseline as appears in Fig. 3, with vLLM being run with default configuration, and percentage differences are reported accordingly. It's important to note that running with chunked prefill disabled reduces P99 ITL since decode requests don't appear in the same batch as prefill requests (since those aren't chunked), but reduces overall throughput for the same reason. To replicate experiments on models for which steering vectors weren't calculated, we inserted random steering vectors of the matching hidden dimension size.

Results. Results are shown in Table 2. We observe an order-of-magnitude lower overheads in xMIx compared to EasySteer across all metrics. These performance benefits stem from EasySteer using the eager vLLM execution with its CPU-driven control flow, a trait shared by all current MI frameworks except xMIx.

Comparing to other platforms. Table 3 provides a comparison against platforms widely used by MI researchers: TransformerLens [Nanda and Bloom, 2022] and NNsight [Fiotto-Kaufman et al., 2024]. We report absolute TTFT, TTT, Mean ITL and P99 ITL under comparable one-layer and all-layer write settings. We didn't perform application comparison as we did for xMIx and EasySteer, since none of the applications were implemented for NNsight and for TransformerLens. Let us note, that TransformerLens isn't vLLM based, and doesn't support batching, significantly lowering TTT, but achieving much better TTFT.

Memory overhead. xMIx consumes only a few KBs of additional memory beside the memory used by MI applications themselves (both their buffers and code). The memory of the applications is managed by vLLM. The total footprint of all the surveyed applications combined is about a few MBs, constituting a negligible fraction of the total memory used by the model and its activations.

Metric	**Baseline**	xMIx		**NNsight**		**TransformerLens**	
	(vLLM)	One layer	All layers	One layer	All layers	One layer	All layers
Llama-3.1-8B							
Throughput (tok/s, ↑)	7596	7550 (−0.6%)	7389 (−2.7%)	3453 (−54.2%)	3365 (−55.3%)	18.17 (−99.7%)	17.18 (−99.7%)
Mean ITL (ms, ↓)	10.52	10.58 (+0.5%)	10.82 (+2.8%)	30.55 (+187.6%)	31.46 (+196.2%)	54.71 (+420%)	57.90 (+450%)
P99 ITL (ms, ↓)	46.13	46.28 (+0.3%)	46.48 (+0.8%)	37.93 (−17.9%)	39.07 (−15.5%)	55.62 (+20%)	59.87 (+30%)
Mean TTFT (ms, ↓)	5301	5336 (+0.7%)	5434 (+2.5%)	2843 (−46.3%)	2888 (−45.4%)	92.37 (−98.2%)	89.91 (−98%)

Table 3: Absolute performance comparison for a multi and single layer steering configuration across xMIx, NNsight and TransformerLens, anchored by the vLLM baseline, reported for Llama-3.1-8b-Instruct. Signed percentage changes relative to the baseline are shown in parentheses for xMIx only.

6 Conclusion

This paper introduces xMIx, a novel framework for integrating mechanistic interpretability applications into SOTA LLM inference flows. By identifying the common ground to many applications, and analyzing it from the system perspective, we have developed an infrastructure, allowing researchers to apply a wide set of different MI methods, while keeping them compatible to existing optimizations. That way, we have removed a obstacle standing in the way of bringing the insights of MI to real life production grade inference.

References

vllm-lens, 2026. URL https://www.lesswrong.com/posts/3bs27nZQuEcKhXf7q/vllm-lens-fast-interpretability-tooling-that-scales-to. Accessed: 2026-05-10.

Andy Arditi, Oscar Obeso, Aaquib Syed, Daniel Paleka, Nina Panickssery, Wes Gurnee, and Neel Nanda. Refusal in language models is mediated by a single direction. *Advances in Neural Information Processing Systems*, 37:136037–136083, 2024.

Matan Ben-Tov, Mor Geva, and Mahmood Sharif. Universal jailbreak suffixes are strong attention hijackers. *arXiv preprint arXiv:2506.12880*, 2025. arXiv preprint used because no published venue was found.

Runjin Chen, Zhenyu Zhang, Junyuan Hong, Souvik Kundu, and Zhangyang Wang. SEAL: Steerable reasoning calibration of large language models for free. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=klPszYDIRT.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35: 16344–16359, 2022.

EasySteer. Server-level steering with cuda graphs. https://github.com/ZJU-REAL/EasySteer-vllm-v1/pull/3, 2026. Accessed: 2026-06-11.

Misbah Farooq, Varuna De Silva, Rahul Rahulamathavan, and Xiyu Shi. Sentiment steering in large language models via activation vector manipulation. In *2025 25th International Conference on Digital Signal Processing (DSP)*, pages 1–5. IEEE, 2025.

Mohsen Fayyaz, Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Ryan A. Rossi, Trung Bui, Hinrich Schuetze, and Nanyun Peng. Steering MoE LLMs via expert (De)activation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=v5Yl9V8rJs.

Jaden Fiotto-Kaufman, Alexander R. Loftus, Eric Todd, Jannik Brinkmann, Caden Juang, Koyena Pal, Can Rager, Aaron Mueller, Samuel Marks, Arnab Sen Sharma, Francesca Lucchetti, Michael Ripa, Adam Belfki, Nikhil Prakash, Sumeet Multani, Carla Brodley, Arjun Guha, Jonathan Bell, Byron Wallace, and David Bau. NNsight and NDIF: Democratizing access to foundation model internals. *arXiv preprint arXiv:2407.14561*, 2024. arXiv preprint used because no published venue was found.

Jason Gaiser, David Fontaine, Houston Hoffman, Stephen Jones, and Fred Oh. Dynamic control flow in CUDA graphs with conditional nodes. https://developer.nvidia.com/blog/dynamic-control-flow-in-cuda-graphs-with-conditional-nodes/, 2024. NVIDIA Technical Blog. Published 2024-05-10, updated 2025-02-03. Accessed 2026-05-07.

Zhanhao Hu, Julien Piet, Geng Zhao, Jiantao Jiao, and David Wagner. Toxicity detection for free, 2024. URL https://arxiv.org/abs/2405.18822.

Sri Durga Sai Sowmya Kadali and Evangelos E. Papalexakis. Jailbreaking leaves a trace: Understanding and detecting jailbreak attacks from internal representations of large language models. *arXiv preprint arXiv:2602.11495*, 2026. arXiv preprint used because no published venue was found.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. doi: 10.1145/3600006.3613165.

Bruce W. Lee, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Erik Miehling, Pierre Dognin, Manish Nagireddy, and Amit Dhurandhar. Programming refusal with conditional activation steering. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Oi47wc10sm.

Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Inference-time intervention: Eliciting truthful answers from a language model. *Advances in Neural Information Processing Systems*, 36:41451–41530, 2023.

Neel Nanda and Joseph Bloom. Transformerlens. https://github.com/TransformerLensOrg/TransformerLens, 2022. Software library citation from the project README; no archival paper located.

NVIDIA. CUDA programming guide: CUDA graphs. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html, 2026a. Official NVIDIA documentation. Accessed 2026-05-04.

NVIDIA. Handling dynamic patterns. https://docs.nvidia.com/dl-cuda-graph/latest/torch-cuda-graph/handling-dynamic-patterns.html, 2026b. In the official NVIDIA guide *CUDA Graph Best Practice for PyTorch*. Accessed 2026-05-07.

Hadas Orgad, Michael Toker, Zorik Gekhman, Roi Reichart, Idan Szpektor, Hadas Kotek, and Yonatan Belinkov. LLMs know more than they show: On the intrinsic representation of LLM hallucinations. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=KRnsX5Em3W.

Hadas Orgad, Fazl Barez, Tal Haklay, Isabelle Lee, Marius Mosbach, Anja Reusch, Naomi Saphra, Byron C. Wallace, Sarah Wiegreffe, Eric Wong, Ian Tenney, and Mor Geva. Interpretability can be actionable. https://actionable-interpretability-guide.github.io/paper.pdf, 2026. Position paper hosted on the project website. The companion page lists this citation. Accessed 2026-05-07.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Turner. Steering llama 2 via contrastive activation addition. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15504–15522, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.828. URL https://aclanthology.org/2024.acl-long.828/.

Marco Scialanga, Thibault Laugel, Vincent Grari, and Marcin Detyniecki. SAKE: Steering activations for knowledge editing. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15966–15978, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.777. URL https://aclanthology.org/2025.acl-long.777/.

ShareGPT. Sharegpt vicuna unfiltered dataset. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2026. Accessed: 2026-06-08.

Asa Cooper Stickland, Alexander Lyzhov, Jacob Pfau, Salsabila Mahdi, and Samuel R. Bowman. Steering without side effects: Improving post-deployment control of language models. In *Neurips Safe Generative AI Workshop 2024*, 2024. URL https://openreview.net/forum?id=tfXIZ8P4ZU.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19. Association for Computing Machinery, 2019. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J Vazquez, Ulisse Mini, and Monte MacDiarmid. Steering language models with activation engineering, 2025. URL https://openreview.net/forum?id=2XBPdPIcFK.

vLLM Project. Benchmark sweep: Serve. https://docs.vllm.ai/en/latest/cli/bench/sweep/serve, 2026a. Accessed: 2026-06-08.

vLLM Project. vLLM KernelConfig api reference. https://docs.vllm.ai/en/stable/api/vllm/config/kernel/, 2026b. Official vLLM documentation. Documents Triton-based fused MoE kernels as a supported backend. Accessed 2026-05-05.

Mengru Wang, Xingyu Chen, Yue Wang, Zhiwei He, Jiahao Xu, Tian Liang, Qiuzhi Liu, Yunzhi Yao, Wenxuan Wang, Ruotian Ma, Haitao Mi, Ningyu Zhang, Zhaopeng Tu, Xiaolong Li, and Dong Yu. Two experts are all you need for steering thinking: Reinforcing cognitive effort in moe reasoning models without additional training. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=x7fCiuCCAu.

Peng Wang, Ningyu Zhang, Bozhong Tian, Zekun Xi, Yunzhi Yao, Ziwen Xu, Mengru Wang, Shengyu Mao, Xiaohan Wang, Siyuan Cheng, Kangwei Liu, Yuansheng Ni, Guozhou Zheng, and Huajun Chen. EasyEdit: An easy-to-use knowledge editing framework for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 82–93, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-demos.9. URL https://aclanthology.org/2024.acl-demos.9/.

Weixuan Wang, Minghao Wu, Barry Haddow, and Alexandra Birch. Bridging the language gaps in large language models with inference-time cross-lingual intervention. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5418–5433, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.270. URL https://aclanthology.org/2025.acl-long.270/.

Weixuan Wang, JINGYUAN YANG, and Wei Peng. Semantics-adaptive activation intervention for LLMs via dynamic steering vectors. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=8WQ7VTfPTl.

Haolei Xu, Xinyu Mei, Yuchen Yan, Rui Zhou, Wenqi Zhang, Weiming Lu, Yueting Zhuang, and Yongliang Shen. Easysteer: A unified framework for high-performance and extensible llm steering, 2025a. URL https://arxiv.org/abs/2509.25175.

Haolei Xu, Xinyu Mei, Yuchen Yan, Rui Zhou, Wenqi Zhang, Weiming Lu, Yueting Zhuang, and Yongliang Shen. Easysteer: A unified framework for high-performance and extensible LLM steering. *arXiv preprint arXiv:2509.25175*, 2025b. arXiv preprint used because no published venue was found.

Ziwen Xu, Shuxun Wang, Kewei Xu, Haoming Xu, Mengru Wang, Xinle Deng, Yunzhi Yao, Guozhou Zheng, Huajun Chen, and Ningyu Zhang. EasyEdit2: An easy-to-use steering framework for editing large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 522–535, Suzhou, China, 2025c. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-demos.38. URL `https://aclanthology.org/2025.emnlp-demos.38/`.

A Full Survey Mapping

This appendix will map each surveyed MI application to the representative application used in Table 1.

#	Surveyed application (paper)	Representative app evaluated	Notes / cluster rationale
1.1	**Single Direction** (Arditi et al. [2024])	**App-1**	Unconditional Multi-layer steering
1.2	Steering Language (Turner et al. [2025])	App-1	Unconditional Multi-layer steering
1.3	Sentiment steering (Farooq et al.) [2025]	App-1	Unconditional Multi-layer steering
1.4	Bridging the Language Gaps in Large Language Models with Inference-Time Cross-Lingual Intervention (Wang et al.) [2025a]	App-1	Unconditional Multi-layer steering
2.1	**SEAL** (Chen et al. [2025])	**App-2**	Token Conditional steering
3.1	**Conditional Refusal** (Lee et al. [2025])	**App-3**	Conditional steering
3.2	Semantics-Adaptive Activation Intervention for LLMs via Dynamic Steering Vectors (Wang et al.) [2025b]	App-3	Conditional steering
4.1	**Hallucination Probe** (Orgad et al. [2025])	**App-4**	Logistic regression used on activations
4.2	Toxicity Detection for Free (Hu et al.) [2024]	App-4	Logistic regression used on activations
5.1	**SAKE** (Scialanga et al. [2025])	**App-5**	
5.2	Steering Llama 2 via Contrastive Activation Addition (Rimsky et al.) [2024]	App-5	Unconditional write on a single layer
6.1	**Side-Effect-Free Steering** (Stickland et al. [2024])	**App-6**	Unconditional Multi-layer steering
7.1	**SteerMoE** (Fayyaz et al. [2026])	**App-7**	MoE Router steering
7.2	Two Experts Are All You Need for Steering Thinking: Reinforcing Cognitive Effort in MoE Reasoning Models Without Additional Training (Wang et al.) [2026]	App-7	MoE Router steering

Table 4: Surveyed-app to representative-app mapping referenced in Section 3.

B Cuda graph support on other frameworks

`EasySteer` started implementing single layer and multi layer steering on Qwen2.5-1.5B-Instruct model with cuda graph support [EasySteer, 2026]. Since `EasySteer` doesn't support any of the applications with cuda graphs, we couldn't provide this comparison in Table 2. We will also notice, that XMIX is the only framework offering support for different applications and primitives (Read, Write, Conditional Write) while preserving full cuda graph compatibility.