MIA: RUNTIME-RECONFIGURABLE ACCESS TO MODEL INTERNALS IN LLM INFERENCE ENGINES

Anonymous authors
Paper under double-blind review

ABSTRACT

As Large Language Models (LLMs) and agents see broader deployment, runtime-time monitoring and intervention become increasingly important. This requires an inference engine to accept diverse requests asking for different layers, tokens, and vectors within one batched step, while meeting the latency targets. However, existing inference engines, which are primarily optimized for high-throughput text generation, are not designed to handle such heterogeneous request efficiently. Recent systems narrow this gap, but each pays for it by trading either throughput, write access, model support, or internal types. Therefore, we present MIA, a plugin for native vLLM server that captures and steers model internals at near-native efficiency. MIA consists of two primary parts, a Scheduler and a Courier. Before each step, the Scheduler decides what each request should read or write based on the batch and turns that decision into a routing entry. The GPU then replays the CUDA graph that reads and writes following those entries with minimal overhead. The Courier moves what the graph produced off the GPU in the background, returning it together with the model's own output. Under open-loop arrivals with requests asking for different layers, MIA generates each output token 2.9–14.1$\times$ faster than existing systems that support per-request hidden state capture during serving from 1 to 16 requests per second.

1 INTRODUCTION

Recent research in understanding Large Language Models (LLMs) has shown that intermediate artifacts (e.g. activations and attention scores) encode information that is not explicitly shown in the output text (Bereska & Gavves, 2024; Templeton et al., 2024; Lindsey et al., 2025). For example, from the safety-monitoring perspective, capturing and analyzing attentions can reveal whether a malicious instruction hidden in the retrieved text has diverted the model from the original task or caused it to violate constraints (Hung et al., 2025; Wang et al., 2026). Model activations can also be modified during inference to affect the final output. For example, adding a steering vector to the residual stream can make a model decline those malicious requests without creating a new model checkpoint (Lee et al., 2025; Arditi et al., 2024). As these operations become practical mechanisms for runtime safety, observability, and behavioral control, inference engines increasingly need to support them as first-class capabilities. However, unlike conventional text generation, these operations introduce request-dependent computation and state that must be dynamically coordinated during inference, while still meeting stringent latency requirements.

Two properties of modern serving engines make this particularly difficult. First, the fast path provides limited flexibility for per-request control. To optimize for the processing efficiency, a decode step is recorded once as a CUDA graph (NVIDIA Corporation, 2026) and replayed afterwards as a fixed sequence of kernels operating on pre-allocated memory addresses. In the meantime, optimizations such as continuous batching (Yu et al., 2022) and chunked prefill (Agrawal et al., 2024) dynamically configure each batch. Capture and steering requirements can therefore vary across batches, which a fixed CUDA graph does not naturally accommodate. Supporting such heterogeneous operations while retaining the high inference performance is thus challenging. Second, activation capture can produce substantial data that seize the GPU memory for inference and require additional offloading scheduling. For a 70B Llama model (Grattafiori et al., 2024), recording hidden states from all 80 layers produces approximately 1.25 MiB for every token. For a request with 8192 tokens, this

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

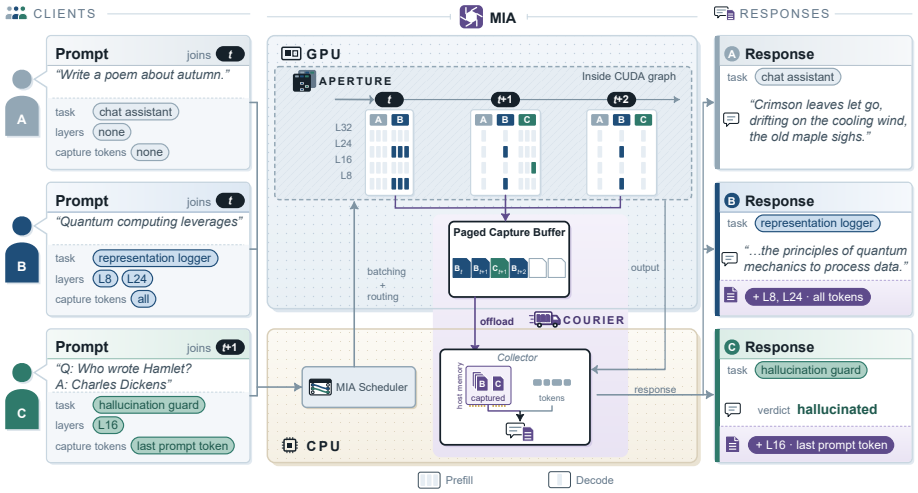


Figure 1: An overview of MIA. The Scheduler (see Fig. 2) resolves each request's selections into routing entries before the step runs. Those entries set the Aperture. The per-step selection marked in the figure decides which layers and tokens the replayed graph acts on. Selected activations enter the paged capture buffer during graph replay. The Courier (see Fig. 3) then offloads and reassembles them for delivery.

amounts to 10.0 GiB for all hidden states. Consequently, the serving system must properly schedule the offloading to host memory so that memory consumption and transfer bandwidth do not affect the inference performance.

Existing works expose parts of the capture and steering interface, but each makes trade-offs among preserving CUDA graph replay (Maniloff, 2026), operating on an unmodified engine (Yu et al., 2026; Xu et al., 2025), and allowing each request to independently specify what to capture (Kwon et al., 2023). No existing implementation satisfies all properties simultaneously. Therefore, we present MIA (**M**odel **I**nternals **A**perture), a plugin that supports per-request internal capture and steering inside the native vLLM server with low inference overhead. MIA's technical novelty lies in coordinating request-specific access and paged capture buffer reuse within a fixed CUDA graph under continuous batching. Concretely, our contributions are:

- We propose **Aperture**, a runtime-configurable access within a fixed execution graph. We express each request's capture and steering requirements as routing data stored at fixed device addresses. GPU operators read these entries during graph replay, allowing access selections and capture destinations to change without interrupting the recorded graph or invoking Python callbacks.
- We develop a well-coordinated scheduling and asynchronous collection. Specifically, we design **Scheduler** and **Courier** that jointly manage access routing and data movement under native serving features such as continuous batching. The Scheduler maps each request's layer and internal selections to the current batch and assigns capture destinations in a bounded paged buffer. The Courier asynchronously offloads and reassembles captured values by request, releasing pages after collection completes. Together, they guarantee accurate and efficient model internal operations on a production-level inference engine.
- We implement MIA as a plugin to an unmodified vLLM engine with full CUDA graph replay. Under Poisson arrivals of ShareGPT conversations (anon8231489123, 2023), capturing all 32 layers at every token only increases the median time per output token (TPOT) by 4.9–8.4% under 32 requests per second (RPS) on Llama-3.1-8B (Grattafiori et al., 2024). Steering every layer on Qwen2.5-1.5B-Instruct (Yang et al., 2025) adds 3.7–5.6%. While serving requests that ask for different layers, MIA generates each output token at least 2.9–14.1× faster than existing systems that support per-request hidden state capture from 1 to 16 RPS on Llama-3.1-8B.

Figure 1 provides an overview of MIA, showing how the Scheduler configures Aperture for each batch and the Courier collects the captured artifacts.

2 BACKGROUND & RELATED WORKS

2.1 ACCESSING MODEL INTERNALS

In decoder-only transformers, the residual stream carries token representations between layers, with each layer's updated representation forming its hidden state. Attention queries and keys determine attention scores (Vaswani et al., 2017), but fused attention kernels do not materialize these scores in GPU memory (Ye et al., 2025). Capturing queries and keys allows the scores to be reconstructed for analysis such as tracking prompt injection (Hung et al., 2025).

To facilitate test-time alignment, activation steering modifies hidden states during the forward pass to influence the model output without changing model weights. A steering vector can be derived from contrasting activations of prompt pairs that differ primarily in a specific behavior (Turner et al., 2023; Rimsky et al., 2024). By adding the vector to the residual stream at a chosen layer with a scalar coefficient controlling the strength, the outputs are expected to move toward the desirable behavior (Zou et al., 2023; Lee et al., 2025; Arditi et al., 2024; Li et al., 2023).

MIA implements capture and steering through operators, defined here as GPU operations incorporated into the model's forward pass. Capture operators copy selected hidden states or query and key vectors into pre-allocated GPU buffers. Steering operators add scaled vectors to selected hidden states. The operators remain fixed across steps, while their selections and parameters can vary.

2.2 EXISTING TOOLS FOR CAPTURE OR STEERING MODEL INTERNALS

vLLM records a decode step once as a CUDA graph (NVIDIA Corporation, 2026) and then replays the recorded sequence of GPU operations. CUDA graph replay requires the operation sequence and the memory addresses referenced by those operations to remain fixed across replays. The feature reduces overhead on GPU–CPU interaction, but it prevents the engine from inserting self-defined operators or changing control flow after the graph is captured. A tool that needs to run its own Python code at each layer can therefore obtain that access only by turning graph replay off and running the model in eager mode. We report existing works' capabilities in Table 2 in Appendix A.

TransformerLens (Nanda & Bloom, 2022) and pyvene (Wu et al., 2024) target offline model interpretability analysis. TransformerLens re-implements the model forward pass to expose model internals such as attention scores, while pyvene wraps a standard model according to a user-specified intervention configuration. Because both control the forward pass over a fixed batch, they expose a broader set of internal states, including attention heads, attention weights, and gradients. However, neither system includes a serving engine for production-level usage.

vLLM-Lens (Cooney et al., 2026) and NNsight (Fiotto-Kaufman et al., 2025) provide per-request activation access inside vLLM. vLLM-Lens installs callbacks at decoder layers. NNsight sends each request's intervention specification to the workers. Both support per-request reads and writes under continuous batching, but also require eager execution to invoke their Python callbacks. Eager execution returns every kernel launch to the CPU at each decode step, causing every request on the server to pay that GPU–CPU interaction overhead even when it requests neither capture nor steering.

For high-throughput server systems, some works run with the vLLM accelerations but support only capture. DMI-Lib (Yu et al., 2026) registers capture operations in the model computation. Native extraction in vLLM (Kwon et al., 2023) reuses the Eagle speculative decoding machinery (Li et al., 2025). It runs a draft model whose attention layers compute nothing and instead write the verifier's per-layer hidden states into key-value cache pages. Glassbox (Maniloff, 2026) replaces the attention backend and reduces attention data to summary features, but operates only under a non-CUDA graph mode. None of these systems supports steering or per-request selection. Specifically, DMI-Lib captures a selected value type at every layer and supports limited model families; native extraction fixes the captured layer set when the engine starts; and Glassbox serves single request at a time and does not allow continuous batching.

On the steering side, Hotwire-vllm (Fajmanová, 2026) inserts steering operations before compilation. Steering vectors stay at fixed addresses and are selected per request. However, it allocates one slot per distinct vector, layer, and strength combination without eviction. A single request steering every layer exhausts its default budget. The vLLM observation plugin (Community contributors,

2026) installs forward hooks for prefill only, removes them before decode so that decode keeps its CUDA graphs for normal inference.

Two existing systems support both capture and steering under production serving. vLLM-Hook (Ko & Chen, 2026) attaches to an unmodified vLLM through its plugin interface, but relies on Python callbacks that require disabling CUDA graph replay. EasySteer (Xu et al., 2025) keeps graph compatible capture and steering through persistent device tables, but ships as a modified vLLM engine, and most of its steering algorithms use piecewise rather than full graph replay. MIA supports request-specific capture and steering within full CUDA graph replay on an unmodified vLLM engine. Its routing interface maps each request's reads and writes onto the current batch, while a paged capture buffer retains captured values until asynchronous collection completes. This design allows both access selections and capture destinations to change between steps without changing the recorded graph, while bounding capture memory and parallelizing collection and inference loads.

3 METHODOLOGY

MIA coordinates model internal access through a *Scheduler* that prepares per-step routing entries and a *Courier* that asynchronously collects and delivers captured artifacts. Section 3.1 describes how operators and buffers are installed before compilation and graph capture. Section 3.2 details the Scheduler's routing procedure. Section 3.3 explains the Courier's collection and delivery pipeline.

3.1 SYSTEM SETUP

The two challenges raised in Section 1 translate into two requirements for preparing a serving engine. First, a replayed CUDA graph (NVIDIA Corporation, 2026) can only repeat fixed actions, including executing predefined operators and accessing fixed memory addresses. The capture and steering operators and all buffers they access must therefore be in place before vLLM compiles the model and records its graphs. Second, captured data compete with the key-value cache for GPU memory, so MIA must bound its GPU memory reservation in advance to avoid memory overflow and promptly offload captured data to free space for subsequent requests.

Before vLLM starts serving, MIA configures the engine in four steps to address these two requirements. First, MIA inserts the capture or steering operators for the configured task at every decoder layer and registers them with the PyTorch compiler, ensuring that every CUDA graph vLLM records later contains these operators. Second, MIA allocates every buffer accessed by these operators at a fixed device address. Because the recorded operators cannot change, each request specifies what to capture or steer through entries in a routing buffer. The buffer has L rows, one per decoder layer, and N columns, one per token in the largest batch that an engine step can hold. A capture engine also allocates a paged capture buffer for the selected hidden states or queries and keys. For steering, MIA allocates a table of steering vectors. When a vector is used for the first time, MIA copies it into an existing slot so that the table remains at a fixed address. Allocating these buffers at startup bounds their GPU memory use and avoids memory overflow.

The last two steps then prepare the engine for execution. In the third step, MIA initializes the routing buffer so that synthetic warm-up inputs produce no artifacts and receive no edits. Capture entries point to reserved discard rows whose contents are excluded from collection, while steering entries use zero strength and leave the residual stream unchanged. Lastly, vLLM compiles and warms up the instrumented model and records its CUDA graphs. Once the inference serving begins, the operators and buffers remain fixed. When a request arrives, the Scheduler (Section 3.2) only rewrites routing entries, while the Courier (Section 3.3) only drains the paged capture buffer.

3.2 THE SCHEDULER

With the operators and buffers prepared, the Scheduler decides how they are used in each inference step. Graph replay never re-enters Python to resolve request-specific access so that every decision has to be written into the routing plane before the forward pass begins. Figure 2 presents an example about capture-only batch through this path.

The Scheduler begins at stage 1, building the entries by resolving each request's layer, phase, and token-position selections against the runner's token layout. In Figure 2, request A asks for layer 16

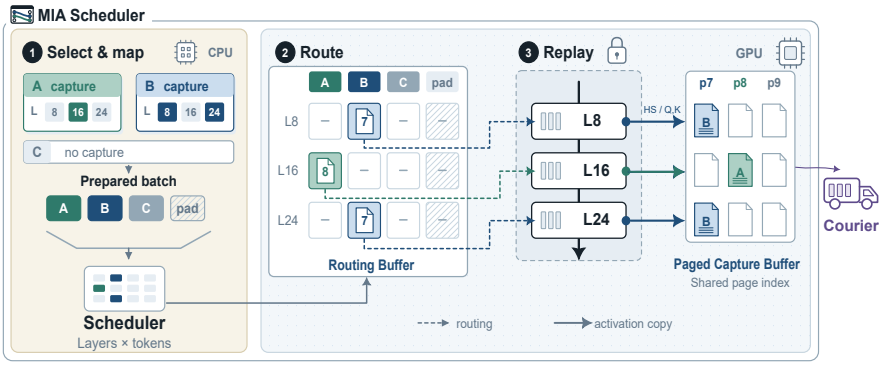


Figure 2: Scheduler routing for one replayed decode step in capture. The Scheduler resolves each request's selections against the prepared batch (1) and publishes one entry per token and layer to the device (2). Operators read the entries during unchanged replay (3) and write each selected activation into a page that stays reserved until the Courier releases it (4).

and request B asks for layers 8 and 24. The mapping also covers the lanes that asked for nothing such as request C, representing a discard destination or carry zero strength. Covering them is a correctness requirement because a replayed step reads every lane it was recorded with. For n real tokens and a largest captured graph of G token lanes, the routing path therefore covers $\min(N, \max(n, G))$ columns, which prevents a padded lane from retaining a stale destination.

Stage 2 transfers the routing entries to the GPU side. During replay, the CUDA graph reads routing information only from the routing buffer at its fixed device address. The entries prepared in stage 1 therefore take effect after they are copied into the buffer. Each capture request is assigned a page index in the paged capture buffer that stores its captured artifacts at the current token. For example, request A is assigned at page 8 and request B is assigned at page 7 to store their current token's activations. The Scheduler writes these entries into a pinned host buffer and copies them asynchronously to the GPU.

During replay at stage 3, the capture operators read the routing buffer and write captured model internals into the paged capture buffer. A captured token's reserved page index is shared across the layers. Request B therefore occupies one page at both of its layers, whereas request A occupies a different page at its single layer. MIA also has a small optimization for capturing the hidden state. When computing the hidden state, vLLM keeps each layer's output and the residual as two separate tensors. This senario costs one kernel for summarizing the layer's output and residual to form the hidden state, and one kernel to store it on the paged capture buffer. MIA therefore fuses these two operations into a single kernel to directly write the sum to its destination.

The same routing interface also controls steering. Each active entry selects a resident steering vector and applies an in-place update to the residual stream. Fixed-strength steering reads its coefficient directly from the routing entry, while activation-dependent steering computes the coefficient from the current residual and performs the projection and addition in one kernel. Inactive entries use zero strength and leave the residual unchanged. Because steering modifies activations in place and writes nothing to the paged capture buffer, the request requires no later collection after the step. Appendix C.6 presents the performance gain brought by capture and steering kernels.

After replay, the captured values remain in the paged capture buffer and wait for the Courier to collect and offload (Section 3.3). Because a page records no request identity, the Scheduler hands the Courier a small record of metadata including which pages the step filled and which request and layers each one belongs to. Later steps take other free pages for new steps. When the buffer is full the engine waits rather than overwrite uncollected data.

3.3 The Courier

The Courier's main job is to offload the collected data in the paged capture buffer and convert them to per-request outputs. It runs on a separate thread, so the inference loop only needs to hand off completed capture pages. Each request fragment must be retained until the offload is completed.

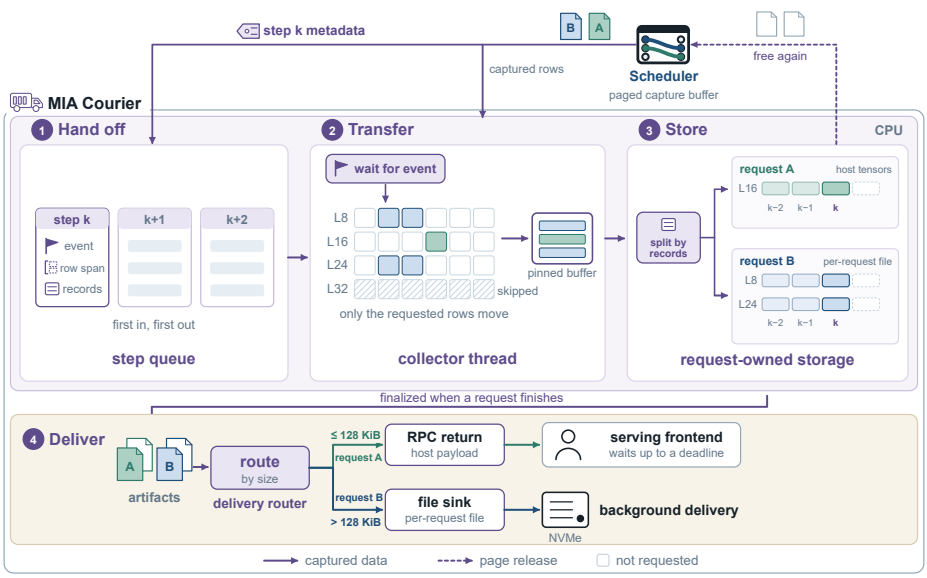


Figure 3: Courier is responsible for collection and delivery for captured data. The engine thread hands the collector an entry holding the step's event, row span and records, while the captured data stays in the GPU buffer (1). A copy stream waits on the event and moves only the requested rows (2), which are split by request using the records and appended to storage each request owns before the pages are freed (3). When a request finishes, a small artifact returns by vLLM's native remote procedure call, a large artifact by a per-request file (4).

Figure 3 shows how the captured data described above in Figure 2 move through the remaining path by the Courier.

Stage 1 begins when the metadata sent by the Scheduler reaches the Courier's step queue. The metadata contains three items. First, a CUDA event recorded on the compute stream after the forward pass indicates when the capture writes are complete. This synchronization is necessary because the engine thread may return before the GPU finishes executing the step, so the collector waits on the event before accessing the captured data. Second, the metadata contains the span of rows in the paged capture buffer written by the step. Third, it includes the Scheduler's mapping records that associate those rows with the corresponding requests and layers. Steps without capture produce no queue entry and entries in the step queue are processed in arrival order.

Stage 2 turns each step queue entry into an actual transfer, offloading data from the GPU to the host memory. The collector processes queue entries in order and waits for each entry's CUDA event before reading its rows. The transfer runs on a dedicated copy stream, allowing the compute stream to continue with serving steps. The collector copies only the rows requested by each layer. It merges the requested span of rows for that layer and skips the transfer entirely if no request selects it. The selected rows are copied into reusable pinned host buffers for stage 3.

After obtaining the transferred rows, stage 3 handles the transferred rows by preserving their request ownership. Using the records in the metadata, the collector separates each step's rows by request and appends them to storage owned by that request. A request's artifact therefore grows across its prefill chunks and the autoregressive decode steps. After a step's rows are stored, the collector advances the drain cursor and releases the corresponding capture pages for reuse.

Finally, Stage 4 determines how the completed artifact is returned. When a request finishes, the engine thread places a completion marker on the same step queue as in stage 1. Since the collector processes entries in order, reaching this marker guarantees that all earlier captures from the request have been transferred and stored. MIA chooses the delivery path at admission based on the estimated artifact size from the prompt length, maximum output tokens, and selected layers. Artifacts predicted to be under 128 KiB remain in host tensors and return through the vLLM frontend's remote procedure call. Larger artifacts are written to a per-request file on the server's NVMe storage to avoid synchronous serialization on the engine loop. The frontend waits for completion up to a configured

Table 1: Median TPOT relative to native vLLM v0.29.0 at the same RPS rate (reference latencies are recorded in the Appendix C.3). A dash marks no run because the GPU is already saturated, and OOM marks out of memory.

System	Engine	Execution	TPOT p50 over native vLLM 0.29				SLO
			1 RPS	4 RPS	8 RPS	16 RPS	knee
Capture, Llama-3.1-8B, all 32 layers, all tokens							
vLLM-Lens, HS	vLLM 0.21	eager	16.8×	62.9×	–	–	<1
Observation Plugin, HS	PR on 0.23	eager	+53.9%	+84.4%	+67.8%	4.2×	8
vLLM-Hook, HS	vLLM 0.21	eager	7.6×	36.3×	OOM	–	<1
vLLM native, HS	vLLM 0.21	piecewise	12.3×	27.7×	19.1×	–	<1
DMI-Lib, HS	fork of 0.11	FULL graph	+6.2%	+6.8%	+7.1%	+14.2%	**32**
MIA, HS (ours)	vLLM 0.29	FULL graph	**+4.9%**	**+6.1%**	**+6.4%**	**+5.8%**	**32**
vLLM-Hook, QK	vLLM 0.21	eager	OOM	–	–	–	<1
DMI-Lib, QK	fork of 0.11	FULL graph	+8.2%	+9.7%	+10.3%	+16.8%	**32**
MIA, QK (ours)	vLLM 0.29	FULL graph	**+6.7%**	**+8.3%**	**+8.4%**	**+7.9%**	**32**
Steering, Qwen2.5-1.5B-Instruct, all 28 layers							
vLLM-Lens	vLLM 0.21	eager	42.7×	203.0×	–	–	<1
vLLM-Hook	vLLM 0.21	eager	4.6×	5.0×	5.4×	10.3×	16
Hotwire-vllm	vLLM 0.21	FULL graph	+9.1%	+18.6%	+87.8%	68.7×	8
EasySteer	fork of 0.29	FULL graph	+10.1%	+10.0%	+4.8%	+6.9%	**64**
MIA (ours)	vLLM 0.29	FULL graph	**+3.9%**	**+4.1%**	**+3.7%**	**+5.6%**	**64**

deadline before reporting a timeout. Since the engine thread only enqueues each step for collection and does not wait for collection to finish, the Courier can process captured data asynchronously in parallel with subsequent inference steps, as long as free capture pages remain available.

4 Evaluation

We evaluate MIA on NVIDIA H100 80GB GPUs (Choquette, 2023), with requests served via HTTP. The baseline comparisons use a single GPU, while the Llama-3.1-70B (Grattafiori et al., 2024) experiments use four GPUs with tensor parallelism. MIA runs as a plugin on native vLLM v0.29.0 with the V2 runner and full CUDA graph capture. Unless otherwise specified, capture uses 4 GiB paged buffer drained to node-local NVMe. Baselines run on their supported vLLM versions, listed in Table 2 in Appendix A. All models use bfloat16 weights with GPU memory utilization set to 90% for Llama-3.1-8B (Grattafiori et al., 2024), Qwen2.5-1.5B-Instruct (Yang et al., 2025), and Llama-3.1-70B.

Requests are drawn from a fixed pool of 512 ShareGPT conversations (anon8231489123, 2023) using a fixed seed, with one human turn and its reply per conversation. Prompts average 186 tokens and replies average 299 tokens for capture, versus 187 and 297 tokens, respectively, for steering. Appendix C.1 explains the arrival schedule and the equal-decode control. A request is considered completed when all data are in the host memory system and ready to be returned.

We report median time to first token (TTFT) and time per output token (TPOT), with overhead measured relative to native vLLM v0.29.0 at the same offered rate. We use the 99th percentile of each metric to identify the highest offered rate that satisfies the service-level objectives (SLOs) of 500 ms TTFT and 50 ms TPOT, so that at most 1% of requests exceed either bound. Reported median latencies are aggregated by taking the median across five independent runs.

4.1 Serving Cost under Production Load

We first compare serving overhead under homogeneous access requirements in Table 1, with every request using the same layer and token coverage. Hidden state (HS) capture records the residual stream at every layer for all prompt and generated tokens. Query-and-key (QK) capture uses the

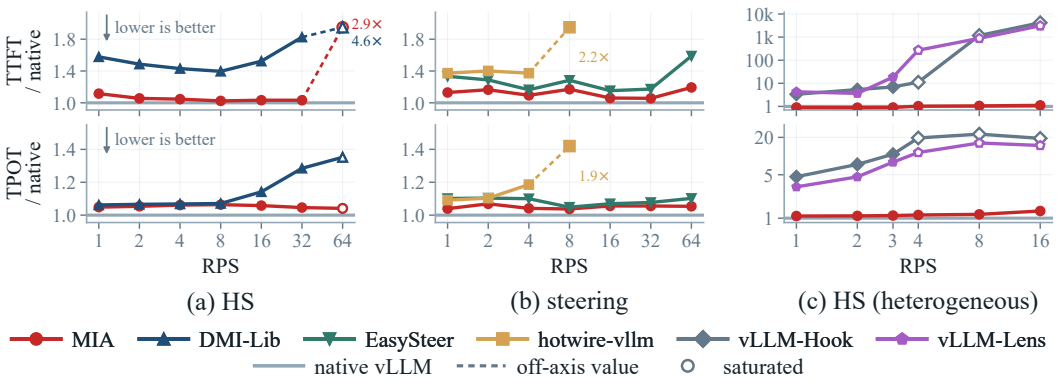


Figure 4: Comparing TTFT and TPOT as a multiple of native vLLM v0.29.0 across same RPS. (a) for hidden states workload and (b) for steering workload, among the systems that keep CUDA graph replay. (c) for hidden states workload, among the systems that allow a different per-request requirement batching at runtime.

same coverage and records all query and key heads. Appendix C.2 derives the per-request capture volumes. Steering applies a unit-norm additive vector at every layer during both prefill and decode, with identical vectors and strengths across systems.

Across the offered rates reported in Table 1, MIA adds 4.9–6.4% median TPOT overhead for HS capture, 6.7–8.4% for QK capture, and 3.7–5.6% for steering. It achieves the lowest median TPOT at every listed rate and matches native vLLM's SLO knees of 32 RPS for capture and 64 RPS for steering. DMI-Lib matches the capture knee, while EasySteer matches the steering knee.

Native hidden state extraction, vLLM-Hook, and vLLM-Lens exhibit approximately 4–203× the reference TPOT. Engine version and execution mode alone do not explain these gaps. On Llama-3.1-8B, native vLLM v0.21.0 in eager mode has a decode time of 9.2 ms, compared with 6.3 ms for v0.29.0 with full CUDA graph replay. With full replay enabled in both versions, their decode times differ by less than 1%. These controls suggest substantial additional overhead from internal-access processing, with queueing further increasing latency at high load. vLLM-Hook exhausts GPU memory at 8 RPS for HS capture and at 1 RPS for QK capture because captured tensors remain on the GPU until requests finish. Other systems encounter excessive queueing as offered load exceeds serving capacity. We omit higher rates once a system can no longer sustain the load.

Figure 4 compares TTFT and TPOT for systems that preserve CUDA graph replay. MIA's overhead includes executing access operators and, when capture is enabled, transferring and storing the data. The fixed per-step cost of operator execution is amortized across larger batches, whereas collection work grows with capture volume. We defer QK capture result to Appendix C.4 due to page limit. Appendix C.5 separates idle instrumentation overhead from the additional cost of active capture.

Similarly to MIA, DMI-Lib executes capture operators within CUDA graphs and offloads captured data asynchronously. However, DMI-Lib copies captured values into its ring buffer in a separate operation, whereas MIA writes directly into its paged buffer and fuses residual reconstruction with the HS capture write, avoiding an additional intermediate-tensor write and read. Appendix C.6 compares MIA's fused kernels with equivalent PyTorch operations, showing larger absolute savings as the number of token rows per step increases. This is consistent with MIA's larger TTFT advantage during prefill, when steps process more tokens. At 16 RPS, MIA adds 5.8% TPOT overhead for HS capture and 7.9% for QK capture, compared with DMI-Lib's 14.2% and 16.8%, respectively.

The steering implementations differ in vector transfer and per-step routing. EasySteer transfers vectors at request admission and retains them in device tables. Hotwire-vllm issues a separate device write for each request and layer at every step. MIA instead packs the step's routing entries into one asynchronous transfer, with steering vectors held in a separate resident table. This avoids a growing number of small device transfers as batch size and layer coverage increase. At 16 RPS, MIA adds 5.6% TPOT overhead, compared with 6.9% for EasySteer, while Hotwire-vllm reaches 68.7× the reference TPOT.

4.2 Heterogeneous Serving Load

After evaluating performance under homogeneous load, we evaluate MIA under heterogeneous batching, where each batch contains requests that capture different layers. This experiment measures MIA under a more realistic mix of per-request capture configurations. Each request captures HS from a randomly selected number of layers, ranging from 0 to 32, in Llama-3.1-8B. We compare MIA with vLLM-Hook and vLLM-Lens, which also support per-request capture at runtime.

Figure 4(c) shows that, from 1 to 16 RPS, MIA generates each output token 4.3–19.6$\times$ faster than vLLM-Hook and 2.9–14.1$\times$ faster than vLLM-Lens. Two costs account for this gap. First, both systems force vLLM to run in eager mode. Second, and more importantly, both capture through Python operators that execute on the engine thread for every requested layer of every request at every step. Their capture operators read each request's token range from a GPU tensor, which blocks the CPU until the GPU completes all queued work. These stalls add roughly 50 μs of step time per captured layer for each request, forcing all other requests in the batch to wait. MIA instead resolves the entire step's capture requests into routing entries before the forward pass and moves the captured rows off the engine thread.

4.3 Scaling to Larger Models with Tensor Parallelism

As we have demonstrated MIA's advantage over baselines in the above sections, here we evaluate whether MIA retains low overhead for larger models (e.g. Llama-3.1-70B) that use tensor parallelism (Shoeybi et al., 2020). We repeat the three workloads on the 70B model across four GPUs and compare with the 8B single-GPU results shown in Section 4.1, examining how increasing computation and operator placement across ranks affect the overhead. Results are reported relative to each model's own native vLLM v0.29.0 baseline in Appendix C.7. At rates up to 8 RPS, both HS and QK capture incur less than 10% TPOT overhead on both models. Steering overhead remains consistent across the sweep at about 2% for the 8B model and 6% for the 70B model. MIA's overhead depends mainly on layer count and model weight placement rather than on total model size.

HS capture becomes slightly cheaper on the larger deployment across all ranks. Installing capture operators costs 1.1% of decode time instead of the 8B model's 3.3%. Under tensor parallelism, every rank holds the same complete hidden state, so only one rank per layer needs to capture it. Each of the four 70B ranks then executes 20 capture operators per step rather than the 32 on the 8B GPU. Queries and keys, however, are partitioned across ranks by head, and steering must update every copy of the residual stream, so both still require an operator at every layer on every rank. At higher rates, capture overhead grows as more rows must be transferred and stored while the operator installation cost stays fixed. For HS capture, the TPOT overhead rises from 0.42 to 3.24 ms between 1 and 24 RPS on the 70B deployment, compared with the TPOT overhead rise from 0.31 to 0.42 ms on the 8B deployment. The larger deployment follows the same overhead pattern observed on the 8B model before saturation, with operator costs amortized across requests and collection costs scaling with capture volume.

5 Conclusion

We present MIA, a plugin that enables per-request capture and steering of model internals within native vLLM while preserving full CUDA graph replay. Our Aperture mechanism expresses changing access requirements as routing data consumed by fixed graph operators. The Scheduler and Courier coordinate these selections with bounded capture storage and asynchronous collection, allowing inference to proceed alongside artifact offloading. Across the evaluated homogeneous and heterogeneous workloads, MIA achieves lower serving overhead than existing capture and steering systems while retaining native vLLM's serving capacity, demonstrating MIA as a practical foundation for model-internal monitoring and intervention in high-throughput LLM serving.

AI USE STATEMENT

In this work, we used generative AI tools, namely Claude through Claude Code (Anthropic) and ChatGPT and Codex (OpenAI), to assist with implementation. These tools wrote and revised parts of the MIA source code and the profiling harness used for our measurements. We did not use generative AI tools to develop conceptual frameworks, propose or refine hypotheses, design or evaluate the research methodology or experiments, generate synthetic datasets, clean or reformat datasets, translate content, or interpret results. Formulating mathematical claims, writing proofs, and qualitative or thematic data analysis are not applicable to this work. We also used these tools to draft and edit parts of the paper, write scripts for generating figures and plots, and search for, identify, and summarize related literature. We reviewed all AI-assisted work. All AI-written code was inspected by the authors and tested for correctness, and every number reported in this paper is traceable to the log of the run that produced it. The authors also rewrote or edited every AI-drafted sentence. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We release the source code of MIA in an anonymized repository at `https://github.com/anonymouseTomCat/MIA_ICLR.git`. Sections 3.1, 3.2 and 3.3 describe the design that the code implements, and Section 4 and Appendix C.1 describe the experimental settings.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation*, pp. 117–134, 2024.

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 4895–4901, 2023.

anon8231489123. ShareGPT_Vicuna_unfiltered dataset. Hugging Face Datasets, 2023. URL `https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered`.

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.

Leonard Bereska and Stratis Gavves. Mechanistic interpretability for AI safety - a review. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL `https://openreview.net/forum?id=ePUVetPKu6`.

Jack Choquette. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, 43(3):9–17, May 2023. ISSN 0272-1732. doi: 10.1109/MM.2023.3256796.

Community contributors. [RFC]: Observation Plugin for Intercepting & Routing on Activations. vLLM GitHub Issue #36998, March 2026. URL `https://github.com/vllm-project/vllm/issues/36998`.

Alan Cooney, Sid Black, and UK AI Safety Institute. vLLM-Lens: Fast interpretability tooling for large language models. https://github.com/UKGovernmentBEIS/vllm-lens, 2026.

Kateřina Fajmanová. hotwire-vllm: Cuda-graph-safe per-request activation steering for vLLM. `https://github.com/moudrkat/hotwire-vllm`, 2026.

Jaden Fiotto-Kaufman, Alexander Loftus, Eric Todd, Jannik Brinkmann, Koyena Pal, Dmitrii Troitskii, Michael Ripa, Adam Belfki, Can Rager, Caden Juang, et al. Nnsight and ndif: Democratizing access to open-weight foundation model internals. In *International Conference on Learning Representations*, volume 2025, pp. 92337–92370, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Kuo-Han Hung, Ching-Yun Ko, Ambrish Rawat, I-Hsin Chung, Winston H. Hsu, and Pin-Yu Chen. Attention tracker: Detecting prompt injection attacks in LLMs. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 2309–2322. Association for Computational Linguistics, April 2025.

Ching-Yun Ko and Pin-Yu Chen. vLLM hook v0: A plug-in for programming model internals on vllm. *arXiv preprint arXiv:2603.06588*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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*, pp. 611–626, 2023.

Bruce W Lee, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Erik Miehling, Pierre Dognin, Manish Nagireddy, and Amit Dhurandhar. Programming refusal with conditional activation steering. In *International conference on learning representations*, volume 2025, pp. 90960–90985, 2025.

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.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. On the biology of a large language model. *Transformer Circuits Thread*, 2025. URL https://transformer-circuits.pub/2025/attribution-graphs/biology.html.

Diego Maniloff. Glassbox. https://github.com/dmaniloff/glassbox, 2026. Accessed: 2026-09-26.

Neel Nanda and Joseph Bloom. Transformerlens. https://github.com/TransformerLensOrg/TransformerLens, 2022.

NVIDIA Corporation. *CUDA C++ Programming Guide*, 2026. URL https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Release 13.4, Accessed: 2026-09-26.

Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Turner. Steering llama 2 via contrastive activation addition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15504–15522, 2024.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits Thread*, 2024. URL https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html.

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J Vazquez, Ulisse Mini, and Monte MacDiarmid. Steering language models with activation engineering. *arXiv preprint arXiv:2308.10248*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Yanting Wang, Runpeng Geng, Ying Chen, and Jinyuan Jia. Attntrace: Contextual attribution of prompt injection and knowledge corruption. In *2026 IEEE Symposium on Security and Privacy (SP)*, pp. 4320–4338, 2026.

Zhengxuan Wu, Atticus Geiger, Aryaman Arora, Jing Huang, Zheng Wang, Noah D. Goodman, Christopher D. Manning, and Christopher Potts. `pyvene`: A library for understanding and improving PyTorch models via interventions. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 158–165, 2024.

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*, 2025.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report, January 2025. URL `http://arxiv.org/abs/2412.15115`. arXiv:2412.15115 [cs].

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pp. 521–538, 2022.

Nengneng Yu, Sixian Xiong, Yibo Zhao, Wei Wang, and Zaoxing Liu. Enabling performant and flexible model-internal observability for llm inference. *arXiv preprint arXiv:2605.11093*, 2026.

Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, et al. Representation engineering: A top-down approach to ai transparency. *arXiv preprint arXiv:2310.01405*, 2023.

A Related works capabilities

Table 2: Capabilities of the related works.

Method	Capture	Steer	Server	Graph safe capture	Graph safe steer	No fork
TransformerLens	✓	✓	✗	✗	✗	✓
pyvene	✓	✓	✗	✗	✗	✓
NNsight (vLLM backend)	✓	✓	✓	✗	✗	✓
glassbox	✓	✗	✓	✗	✗	✓
DMI-Lib	✓	✗	✓	✓	✗	✗
vLLM native HS	✓	✗	✓	✓	✗	✓
vLLM Observation Plugin	✓	✗	✓	✗	✗	✗
Hotwire-vllm	✗	✓	✓	✗	✓	✓
EasySteer	✓	✓	✓	✓	✓	✗
vLLM-Lens	✓	✓	✓	✗	✗	✓
vLLM-Hook	✓	✓	✓	✗	✗	✓
MIA (ours)	✓	✓	✓	✓	✓	✓

B Limitation of MIA

This section explains the limitation of MIA: pages in the paged capture buffer are released only after stage 3 in the Courier 3.3 stores their contents. If the inference steps on GPU produces captured data faster than the Courier can drain it, the paged capture buffer can become saturated, causing the Courier to become the bottleneck and forcing the inference engine to wait for free pages. Inference performance can therefore becomes limited by the collector's processing speed. Although the Courier removes data transfer and storage from the engine thread, but their throughput still determines how quickly pages are recycled. Section 4.1 investigates this bottleneck and compares its overhead with existing works.

C Evaluation details

In this section, we explain the details of our experimental settings in Section 4.

C.1 Load generation and decode control

We control the offered load using a Poisson arrival process from 1 to 128 RPS, doubling the rate at each step. We evaluate each rate for 30 seconds after a 30-second warm-up. Every request for steering decodes exactly the number of tokens in its recorded reply using greedy sampling, with end-of-sequence (EoS) stopping disabled so that all systems decode the same number of tokens. This control is important for steering because steering changes the output distribution and could otherwise terminate generation before the prescribed reply length. We stop each system's RPS sweep when it can no longer sustain the offered load and report that RPS as its collapse rate.

C.2 Per-Request Capture Volume

The volume one request produces is fixed by the model's geometry and by how many tokens the capture covers. This appendix derives that volume for Llama models (Grattafiori et al., 2024) used in Section 4.

Let L be the number of decoder layers, d the hidden size, H the number of attention heads, H_{kv} the number of key-value heads under grouped-query attention (Ainslie et al., 2023), d_h the head dimension, and b the bytes per element, which is 2 for the bfloat16 weights used throughout. Let $T = P + G$ be the number of tokens a request covers, where P is its prompt length and G the

number of tokens it generates. Hidden state capture records one residual stream row per layer per token, and that row is d values wide, so

$$S_{\mathrm{HS}} = L \cdot d \cdot b \cdot T. \tag{1}$$

Query-and-key capture records two rows per layer per token instead of one, because the quantity an analysis wants is the attention score rather than the query or the key on its own. For a head with query matrix Q, key matrix K and value matrix V, attention computes

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^{\top}}{\sqrt{d_h}}\right) V, \tag{2}$$

where the score matrix is the softmax term (Vaswani et al., 2017). Since vLLM's fused attention kernels (Kwon et al., 2023) in a serving engine never write that term to memory, a request recovers it by capturing the post-rotary Q and K that produced it and recomputing the product after the fact. Capture therefore stores one query row and one key row per layer per token. The query row is $H \cdot d_h$ values wide, giving one vector of d_h values for every attention head. The key row is narrower at $H_{kv} \cdot d_h$ values, because grouped-query attention projects one key head for each group of query heads and the captured tensor is that projection rather than the copies attention broadcasts from it. Therefore

$$S_{\mathrm{QK}} = L \cdot b \cdot T \cdot \big(\underbrace{H \cdot d_h}_{\text{query row}} + \underbrace{H_{kv} \cdot d_h}_{\text{key row}}\big). \tag{3}$$

Tables 3 and 4 evaluate Equations 1 and 3 for Llama-3.1-8B and Llama-3.1-70B (Grattafiori et al., 2024) that for capture workloads at the average request length of the ShareGPT pool. Steering appears in none of them because it edits the residual stream in place and writes no artifact, so its volume is zero at every coverage.

Table 3: Capture volume for Llama-3.1-8B, with $L = 32$, $d = 4096$, $H = 32$, $H_{kv} = 8$, $d_h = 128$ and $b = 2$. The per-request column assumes $T = 485$ tokens, the pool's average of a 186-token prompt and a 299-token reply.

Tensor	Row width per layer	Per token	Per request
Hidden states	4096	256 KiB	121 MiB
Query and key	5120	320 KiB	152 MiB

Table 4: Capture volume for Llama-3.1-70B, with $L = 80$, $d = 8192$, $H = 64$, $H_{kv} = 8$, $d_h = 128$ and $b = 2$. The per-request column assumes the same $T = 485$ tokens as Table 3, so the two are directly comparable.

Tensor	Row width per layer	Per token	Per request
Hidden states	8192	1.25 MiB	606 MiB
Query and key	9216	1.41 MiB	682 MiB

C.3 Native reference latencies

Table 1 reports every system as a ratio against native vLLM v0.29.0 at the same RPS rate. Table 5 gives the absolute latencies behind those ratios, so a relative figure can be read back in milliseconds.

Table 5: Native vLLM v0.29.0 median TPOT at four RPS rate reported in Table 1, and the highest rate at which it meets the SLOs. The capture row uses Llama-3.1-8B and the steering row uses Qwen2.5-1.5B-Instruct.

Model	TPOT p50 (ms)				SLO knee
	1 RPS	4 RPS	8 RPS	16 RPS	
Llama-3.1-8B	6.34	6.50	6.57	6.92	32
Qwen2.5-1.5B-Instruct	2.14	2.18	2.20	2.25	64

C.4 Performance comparison on QK capture

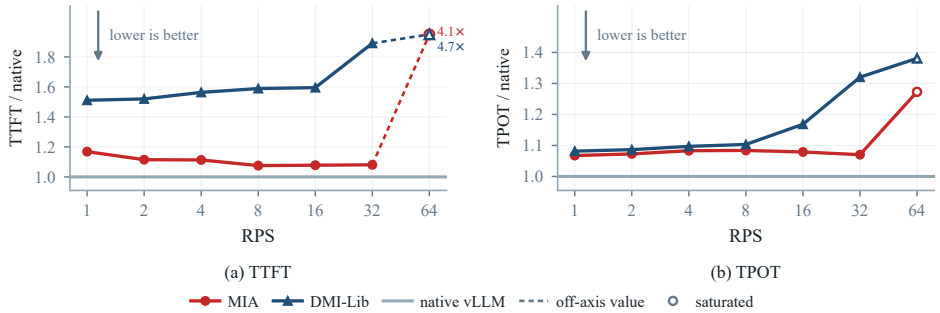


Figure 5: Comparing the systems that keep CUDA graph replay on TTFT and TPOT as a multiple of native vLLM v0.29.0 across same RPS for QK capture workload.

C.5 Overhead Decomposition

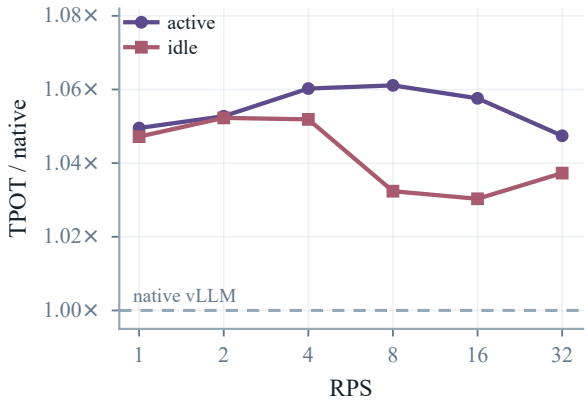


Figure 6: MIA's idle and active tiers comparison on the HS capture workload.

Section 4.1 reports a single overhead value per system, but capture combines two costs with different scaling behaviors. The first comes from keeping the access operators in the replayed graph. It is paid on every forward pass, even when capture is not activated. The second comes from moving captured data from GPU to host memory, incurring data-transfer overhead. We reveal these costs with two experiments: one at the serving system level and one at the kernel level.

As shown in Figure 6, we run an idle MIA with access operators installed but no requests asking for capture, and compare it with the results in Section 4.1. Operator presence alone adds 3.0 to 5.2% overhead to TPOT across sweeps. Moving roughly 100 MiB of residual stream data per request to durable storage adds at most another 2.9%. Most of the cost therefore comes from keeping

the operators present rather than from moving the captured bytes. This is consistent with moving delivery work off the engine thread.

C.6 FUSED KERNELS

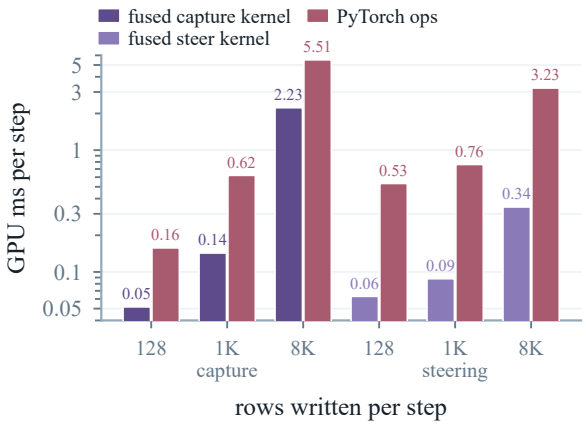
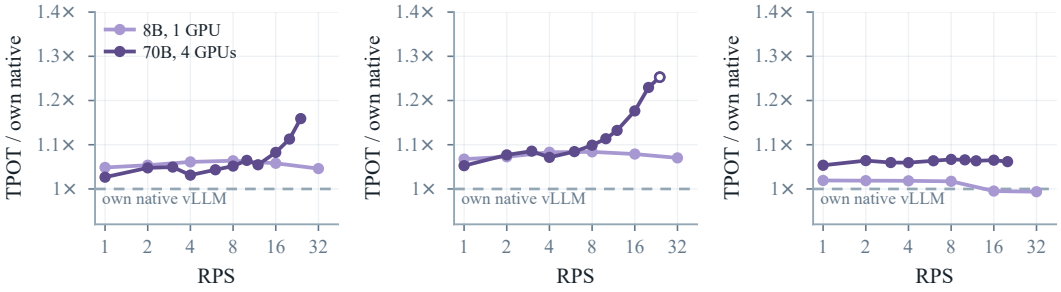


Figure 7: GPU time per step of MIA's two fused kernels against the same work written as PyTorch operations.

MIA uses fused kernels for hidden state capture and steering, as described in Section 3.2. Figure 7 compares these kernels with equivalent PyTorch operations. The effect of kernel optimization on the serving metric depends on the width of the step. The fused capture kernel saves more on wider rows, such as during the prefill stage. However, during the decode stage, the engine admits at most 128 concurrent requests, so the step width remains in the narrowest regime. Fusion speeds up the capture kernel by about 3× and the steering kernel by close to 9×.

C.7 RELATIVE TPOT COMPARISON FOR DIFFERENT MODEL SIZE



(a) HS capture (b) QK capture (c) Steering

Figure 8: Relative TPOT of MIA on Llama-3.1-8B with TP=1 vs. Llama-3.1-70B with TP=4. Each model compares with its own native performance.