eInfer: Unlocking Fine-Grained Tracing for Distributed LLM Inference with eBPF

ABSTRACT

Modern large language model (LLM) inference workloads run on complex, heterogeneous distributed systems spanning CPUs, GPUs, multi-GPU setups, and network interconnects. Existing profiling tools either incur prohibitive overhead, provide limited visibility, or suffer from vendor lock-in, making real-time, fine-grained performance analysis impractical in production environments. We present eInfer, the first eBPF-based system enabling transparent, low-overhead end-to-end tracing of per-request performance across distributed LLM inference pipelines without requiring application modifications. eInfer uniquely correlates events across CPUs, accelerators, processes, and nodes, delivering unified, vendor-agnostic observability that approaches the accuracy of specialized GPU profiling tools. To address the challenges of scalability, dynamic workloads, and instrumentation gaps on accelerators, we design a runtime-adaptive tracing mechanism that maintains comprehensive visibility in real time. Our initial evaluation demonstrates that eInfer delivers precise, low-overhead profiling, enabling critical insights to optimize LLM serving performance in production environments.

1 INTRODUCTION

Observing and understanding the runtime behavior of Large Language Model (LLM) inference systems remains a critical unsolved challenge [4, 30]. Production LLM deployments operate as black boxes where developers lack visibility into how individual requests traverse CPU preprocessing, GPU kernel execution, memory hierarchies, and cross-device communication [27]. This opacity makes it nearly impossible to diagnose performance anomalies, optimize resource allocation, or meet stringent latency SLAs [26]. While operators can measure aggregate metrics like throughput or average latency, they cannot pinpoint why specific requests experience slowdowns, which components cause delays, or how batching decisions impact individual request latencies [13, 15, 21].

The core challenge is that effective LLM inference observation must satisfy multiple stringent requirements that existing tools cannot simultaneously meet. It must be fine-grained, providing per-request visibility across all subsystems to diagnose why individual requests experience variable latency across CPU preprocessing, GPU execution, and interconnect communication [23]. It must be transparent, requiring no modifications to model code or LLM frameworks, as production deployments often involve closed-source components, managed services, or legacy systems where instrumentation is impossible [11]. It must be lightweight, introducing minimal overhead to avoid perturbing the runtime behavior of latency-sensitive systems where even small delays violate SLAs [8, 14, 22]. It must be hardware-agnostic, supporting diverse accelerator vendors and CPU-GPU interconnects, as modern deployments span NVIDIA, AMD, Intel GPUs, and various interconnect technologies. Finally, it must be framework-independent, operating consistently across different inference stacks and serving environments, from PyTorch and TensorFlow to specialized engines such as TensorRT-LLM and vLLM [6].

Traditional profiling approaches fail to meet these requirements. GPU-focused tools like NVIDIA's CUPTI and Nsight Systems provide detailed kernel telemetry but lack CPU-side visibility and request-level context. Framework-specific profilers such as PyTorch Profiler and TensorFlow Profiler require explicit instrumentation and are confined to their respective software stacks. Distributed tracing solutions like Jaeger or application performance monitoring tools depend on code modifications, making them unusable in closed-source or managed environments. Most critically, these tools introduce substantial overhead, often adding tens to hundreds of milliseconds per request, which makes them impractical for production systems where every millisecond impacts user experience.

To break this fundamental tradeoff between visibility and performance, we introduce eInfer, a transparent observability framework that enables live, fine-grained tracing of production LLM inference systems through eBPF (extended Berkeley Packet Filter) [35]. The key insight is that eBPF allows us to dynamically inject monitoring logic directly into the kernel, capturing system behavior without modifying applications or frameworks. By intercepting low-level events such as GPU driver calls via `ioctl`, memory operations, and CPU scheduling decisions, eInfer reconstructs the complete execution timeline of individual inference requests with sub-millisecond overhead. This approach enables continuous monitoring of production systems, providing the visibility needed to debug performance issues as they occur.

An innovation in eInfer is its ability to extract request-level semantics from kernel-level observations. We develop techniques to infer request boundaries from syscall patterns, correlate GPU command submissions with CPU-side batching logic, and track cross-device dependencies without any application cooperation. This enables per-request attribution

even in complex scenarios where multiple batches execute concurrently across multiple GPUs, precisely the conditions where traditional profiling breaks down.

The main contributions of this work are:

- We present eInfer, the first system to enable end-to-end, per-request performance breakdown for LLM inference across CPUs, GPUs, multi-GPU setups, and interconnects using eBPF.
- We enable transparent, low-overhead tracing that accurately correlates per-request performance across multiple components, devices, and distributed nodes—without requiring application code modifications or disrupting production environments.
- We extend eBPF-based tracing to achieve unified, vendor-agnostic observability of CPU and accelerator (e.g., GPU) execution by correlating syscall and driver-level activities, delivering insights comparable to specialized vendor tools like CUPTI.
- We design a scalable, runtime-adaptive tracing mechanism that maintains fine-grained visibility across heterogeneous devices, processes, and distributed nodes in dynamic production environments.

2 MOTIVATION AND CHALLENGES

2.1 Motivation

To understand the limitations of existing profiling tools for LLM inference observation, we conducted a comprehensive evaluation of three representative approaches: PyTorch Profiler (application-level), CUPTI (GPU-specific), and eBPF (system-level). We measured their profiling accuracy, runtime overhead, and practical usability across various LLM models and deployment scenarios.

Our first evaluation focused on PyTorch Profiler's overhead impact on LLM inference latency. As shown in Figure 1, PyTorch Profiler introduces severe performance degradation across all model sizes. The overhead is particularly catastrophic for smaller models: Qwen3-0.6B experiences a 4.67× slowdown, while LLaMA3-8B suffers a 3.16× increase in inference latency. Even as model size grows, the overhead remains substantial—LLaMA2-13B shows a 2.22× slowdown and LLaMA3-70B still incurs a 22% latency increase. This overhead pattern reveals that PyTorch Profiler's instrumentation cost dominates the actual computation time for smaller models, making it entirely impractical for real-time monitoring. Moreover, PyTorch Profiler requires explicit code instrumentation and often necessitates application restarts to collect trace data, introducing significant operational friction in production environments.

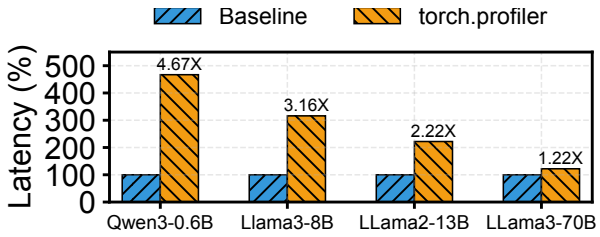
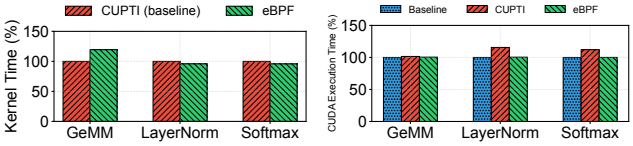


Figure 1: Torch Profiler Latency



(a) Profiling Accuracy **(b) Profiling overhead**

Figure 2: Profiling Results

> **Takeaway 1:** PyTorch Profiler incurs prohibitive overhead and requires intrusive instrumentation, making it unsuitable for real-time, scalable LLM inference profiling in production environments.

We next evaluated the accuracy and overhead of CUPTI and eBPF for GPU kernel profiling. As shown in Figure 2(a), both tools achieve remarkably similar profiling accuracy when measuring kernel execution times for critical LLM operations. Using CUPTI as the ground-truth baseline, we measured kernel runtimes for three representative operations: GeMM (0.185ms), LayerNorm (4.929ms), and Softmax (4.699ms). eBPF reported nearly identical values with deviations of less than 20 microseconds in most cases, demonstrating impressive precision for a general-purpose system tracer. More importantly, as shown in Figure 2(b), both tools maintain negligible runtime overhead. The baseline GeMM operation takes 138.578ms, while CUPTI increases this to 140.744ms (1.56% overhead) and eBPF to 139.281ms (0.51% overhead). This sub-2% overhead stands in stark contrast to PyTorch Profiler, which can inflate execution time by orders of magnitude, making CUPTI and eBPF suitable for production environments.

Beyond accuracy and overhead, we analyzed each tool's capabilities across multiple dimensions critical for distributed LLM inference, as summarized in Table 1. While CUPTI matches eBPF in GPU profiling performance, it remains limited to NVIDIA hardware and cannot observe CPU, memory, or network activity that are essential in distributed deployments. PyTorch Profiler offers some multi-device awareness but suffers from prohibitive overhead and requires intrusive instrumentation. In contrast, eBPF emerges as the only solution combining high accuracy, minimal overhead, and

Feature	CUPTI	eBPF	PyTorch Profiler
Accuracy	◑	●	◐
Overhead	◑	●	○
Device Coverage	◑	●	◐
System Coverage	◑	●	◐
Instrumentation Required	●	●	○
Transparency	◑	●	◐
Hardware Agnostic	○	●	○
Multi-device Awareness	○	●	◑
Ease of Integration	◑	●	◐
Scalability	◐	●	○
Dynamic On/Off at Runtime	○	●	○

Table 1: Comparison of profiling tools for distributed LLM inference. Legend: ● = full/best, ◑ = good/partial, ◐ = limited, ○ = none/poor.

comprehensive system coverage. Its ability to dynamically attach to running systems without code modification provides transparent visibility across CPUs, GPUs (via syscall and driver interfaces), memory operations, and network communication. This unique combination of performance and full-system observability makes eBPF the most practical foundation for understanding complex interactions in modern LLM serving systems that span multiple devices, processes, and nodes.

> **Takeaway 2:** CUPTI and eBPF provide highly accurate, low-overhead profiling, with eBPF uniquely enabling transparent full-system observation across heterogeneous hardware, making it the most practical foundation for distributed LLM inference tracing.

2.2 Challenges in Distributed *eBPF* Tracing

While our motivation study demonstrates that eBPF provides the foundation for LLM inference observation by combining low overhead, system-wide visibility, and hardware independence, building a practical distributed tracing system on top of eBPF introduces several fundamental challenges. These challenges stem from the gap between what eBPF can observe at the kernel level and what developers need to understand about their LLM inference workloads. We identify three critical challenges that must be addressed to realize eInfer vision of transparent, fine-grained distributed LLM tracing.

Challenge 1: End-to-End Event Correlation Across Devices, Processes, and System Layers.

Distributed LLM inference spans a highly heterogeneous and dynamic execution environment, involving CPUs, GPUs, I/O subsystems, and network interfaces, all orchestrated across multiple processes, threads, and sometimes even multiple nodes. A fundamental challenge is achieving precise and consistent correlation of low-level system events across this stack to reconstruct an accurate execution timeline. While eBPF excels at tracing kernel-level events (e.g., syscalls, I/O waits, scheduler decisions), these events lack semantic context about the high-level operations occurring in ML frameworks—such as transformer layer execution, attention kernel launches, or token emission phases. Bridging this semantic gap requires integrating metadata from user-space runtimes (e.g., PyTorch) while preserving low overhead and without modifying application code. Additionally, distributed systems introduce complications like clock drift between nodes, PID/NID reuse, and asynchronous scheduling, all of which can break temporal alignment. Multi-process awareness is further complicated by containerization, dynamic spawning of worker processes, and inter-process communication via shared memory or RDMA. Constructing a global, coherent execution view that maps system-level activities back to model-level abstractions remains an unsolved and technically demanding problem.

Challenge 2: Incomplete Visibility and Instrumentation Gaps on Accelerators.

eBPF is inherently limited in its ability to observe GPU internals due to the closed nature of most GPU drivers and lack of native kernel-level tracepoints for accelerator activities. While it is possible to infer some GPU behavior by observing related syscalls (e.g., `ioctl`, `mmap`, `poll`) or DMA transfers, such inference is often coarse-grained, imprecise, and insufficient for capturing detailed execution characteristics such as kernel launch latency, memory transfer efficiency, or warp divergence. Tools like CUPTI provide these insights but are vendor-specific (NVIDIA-only), intrusive, and incompatible with eBPF's lightweight and hardware-agnostic philosophy. Moreover, integrating CUPTI with eBPF to achieve unified tracing introduces synchronization and data consistency challenges, especially when GPU operations are offloaded asynchronously. As LLM inference workloads increasingly leverage heterogeneous compute—e.g., GPUs, TPUs, and custom accelerators—maintaining visibility into execution on non-standard hardware without compromising eBPF's low overhead and portability becomes a critical limitation. Addressing this challenge requires new techniques for cooperative user-kernel instrumentation or novel abstractions to bridge visibility gaps in accelerator pipelines.

Challenge 3: Scalable Data Collection, Overhead Control, and Runtime Adaptability.

Capturing fine-grained system traces in real time across large-scale distributed environments inevitably generates a massive volume of telemetry data. Without proper filtering, batching, and adaptive sampling, the instrumentation

overhead can quickly overwhelm system resources—leading to probe-induced latency, distorted performance profiles, or even system instability. Although eBPF is designed for low-overhead data collection, practical deployments must still contend with kernel buffer limits, memory pressure, and contention in high-throughput environments. Furthermore, distributed LLM inference workloads are inherently dynamic, with nodes joining or leaving, processes being migrated, models being swapped, and compute resources being rebalanced on the fly. A tracing system must adapt in real time to these changes, maintaining probe coverage, minimizing blind spots, and avoiding stale metadata without requiring system restarts or reconfiguration. This need for dynamic, runtime-aware introspection pushes the limits of current eBPF tooling, which was originally designed for relatively static and localized instrumentation. Finally, once data is collected, centralized aggregation, synchronization, and querying across multiple nodes introduces additional challenges in scalability and fault tolerance—particularly when attempting to reconstruct long execution traces or identify performance regressions at sub-millisecond granularity.

3 DESIGN

The design of our system(shown in Figure 3) is motivated by three fundamental challenges in profiling complex, large-scale distributed applications. **First**, it is essential to capture fine-grained performance metrics across the entire software and hardware stack without requiring any modifications to user code or manual instrumentation. **Second**, meaningful analysis depends on the ability to accurately correlate events across diverse components and devices, even when these operate independently or span multiple abstraction layers. **Third**, the monitoring infrastructure must be able to adapt fluidly to changing workload behaviors and system conditions, all without interfering with normal application execution. Addressing these challenges calls for a cohesive and carefully engineered solution—one that combines lightweight and fine-grained runtime interception, transparent context propagation, and dynamic observability controls to deliver comprehensive insight while remaining unobtrusive and efficient.

3.1 Cross-Layer Event Coordination

To address Challenge 1, which involves end-to-end event correlation across devices, processes, and system layers, we introduce a unified telemetry framework **CoTrace** that tracks inference requests as they traverse the entire system stack. Our framework solves the fundamental problem of correlating low-level kernel events with high-level ML operations by systematically instrumenting each stage of the inference

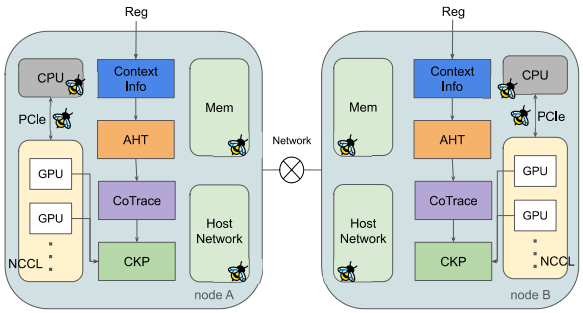


Figure 3: The Architecture of eInfer

pipeline, from initial request arrival through CPU preprocessing, GPU execution, and cross-device communication.

CoTrace follows each inference request through four distinct execution stages, maintaining correlation across all transitions in the Table 2.

Context Propagation Through System Layers. The core innovation in our framework is its ability to maintain request identity as execution crosses abstraction boundaries. When a request enters the system, we assign it a globally unique `trace_id` composed of a timestamp, node identifier, and monotonic counter. This ID is then propagated through:

- **User-to-kernel transitions**: Embedded in syscall arguments via eBPF maps.
- **CPU-to-GPU handoffs**: Injected into CUDA stream metadata.
- **Cross-process boundaries**: Automatically inherited through eBPF tracking of `fork()/clone()` syscalls, where child processes inherit parent's trace context without application awareness.
- **Network communications**: Encoded in NCCL tag fields or RDMA immediate data.

Semantic Enrichment Without Code Modification. Our framework bridges the semantic gap between system events and ML operations through a two-layer approach. At the system level, eBPF programs capture raw events with nanosecond precision. In parallel, a lightweight user-space daemon uses Python introspection hooks (`sys.setprofile`) to detect ML framework calls without modifying application code. These two streams are correlated using the propagated trace ID, allowing us to annotate system events with semantic information.

Distributed Timeline Synchronization. For multi-node inference, our telemetry system implements a hybrid clock synchronization protocol. Each node maintains a local timeline using hardware cycle counters (TSC), while periodic NTP synchronization establishes global time anchors. Cross-node events (e.g., NCCL operations) carry both sender and receiver timestamps, allowing us to compute clock skew and

construct a globally consistent event timeline with microsecond accuracy.

Cross-Layer, Event-Driven Execution Graph. The telemetry pipeline produces a rich execution graph that is temporally aligned and semantically annotated across system layers. Each node in the graph represents a distinct ML operation—such as `LlamaDecoderLayer.forward()`, a KV-cache copy, or a CUDA attention kernel—while edges capture correlated system events, including PCIe transfers, CPU-GPU context switches, and inter-GPU NCCL communications. This cross-layer graph facilitates fine-grained profiling and latency decomposition, enabling precise identification of inference bottlenecks. It also supports detection of unexpected stalls—such as those caused by GPU queue saturation, memory contention, or RDMA backoff—and guides the optimization of scheduling, batching, and resource placement strategies throughout the execution stack.

3.2 Cooperative Kernel Proxies for Accelerator Tracing

To address Challenge 2, which highlights eBPF's limited visibility into accelerator internals, we introduce **Cooperative Kernel Proxies (CKPs)**, a hybrid instrumentation approach that bridges the gap between kernel-level observation and accelerator execution. CKPs solve the fundamental problem that eBPF cannot directly observe GPU kernel launches, memory transfers, or execution details due to closed driver architectures and missing kernel tracepoints.

Lightweight User-Space Instrumentation. CKPs are minimal instrumentation modules embedded within existing user-space components where accelerator information is naturally available, such as PyTorch's CUDA runtime, TensorFlow's GPU executor, or NCCL communication libraries. Unlike heavyweight profiling tools such as CUPTI that require vendor-specific drivers, CKPs extract only essential metadata at points where it is already accessible. This includes kernel launch events (function name, grid dimensions, stream ID), memory operations (transfer size, direction, buffer addresses), synchronization points (stream waits, event records, barrier operations), and resource allocation details (buffer creation, destruction, and sharing across contexts).

The key innovation is how CKPs communicate with eBPF programs without modifying kernel drivers. When a GPU operation occurs, the CKP writes a structured event to a shared memory ring buffer, which eBPF programs then correlate with kernel observations(shown in Figure 8)

Multi-Accelerator Support. The CKP interface is deliberately generic to support heterogeneous accelerators beyond GPUs. Whether tracking TPU operations through XLA runtime, monitoring FPGA execution via OpenCL, or observing custom ASIC behavior through vendor SDKs, the same event schema applies: timestamp, operation type, and minimal metadata. This uniformity allows our framework to trace complex inference pipelines that span multiple accelerator types without vendor-specific modifications to the core tracing infrastructure.

3.3 Adaptive Hierarchical Telemetry

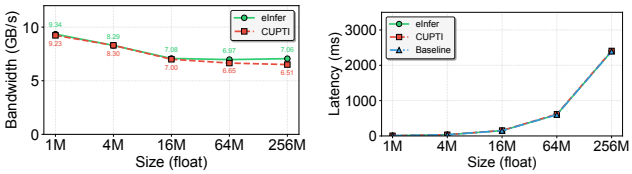
To address Challenge 3, which involves scalable data collection and runtime adaptability in distributed environments, we introduce **Adaptive Hierarchical Telemetry(AHT)** that dynamically balances trace detail with system overhead. This framework solves the fundamental problem that fine-grained tracing across hundreds of GPUs can generate overwhelming data volumes while system conditions constantly change due to workload migration, node failures, and resource rebalancing.

Hierarchical Data Reduction and Dynamic Control. Our framework implements a three-tier architecture where eBPF programs at the kernel level perform aggressive local filtering and event coalescing, node-level aggregators apply pattern detection and compression, and a cluster-level layer merges traces while maintaining temporal consistency. High-frequency events like scheduler context switches are sampled probabilistically, while critical events like GPU kernel launches are always captured. The kernel layer coalesces repetitive operations into compact representations, reducing data volume at the source.

The system employs dynamic overhead control through feedback loops at each aggregator. By monitoring local resource consumption (CPU, memory, network), aggregators automatically adjust sampling rates, aggregation windows, and active tracepoints to stay within overhead budgets. For example, during peak inference load, the system might reduce CPU scheduling traces while maintaining full GPU kernel visibility, ensuring that the most important performance data is preserved without overwhelming system resources.

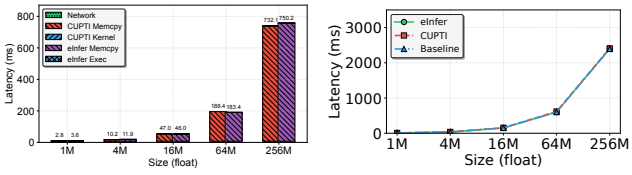
Distributed Coordination and Streaming Pipeline. To handle the dynamic nature of distributed LLM deployments, telemetry aggregators implement coordination protocol where they periodically exchange lightweight metadata about trace sessions and configurations. When workloads migrate or nodes join the cluster, they automatically inherit appropriate tracing settings without manual intervention. This decentralized approach avoids single points of failure while maintaining global consistency in trace collection across the entire cluster.

All telemetry flows through asynchronous, non-blocking pipelines using lock-free ring buffers and memory-mapped overflow files. This design handles burst scenarios gracefully, such as simultaneous launches of thousands of GPU kernels during model initialization, by applying backpressure rather



(a) Profiling Accuracy (b) Profiling overhead

Figure 4: Accuracy and Overhead of Only Enabling Data Transfer Hook



(a) Profiling Accuracy (b) Profiling overhead

Figure 5: Accuracy and Overhead of Multiple Hooks

than dropping events. Stream processors prioritize recent events over historical data when buffers approach capacity, ensuring that the system maintains real-time visibility even under extreme load conditions.

4 EVALUATION

Testbeds. All our experiments were conducted on systems equipped with NVIDIA A6000 GPUs with 48GB of HBM. In addition, each system features 1TB of DRAM, and the GPUs are connected to the host via PCIe Gen 4.0x16.

To assess the profiling accuracy and overhead of memory transfer tracing, we first enable only the memory copy hook between CPU and GPU and systematically vary the transfer size. As shown in Figures 4(a) and 4(b), eInfer achieves results comparable to CUPTI,demonstrating its effectiveness as a lightweight and efficient alternative for GPU memory transfer profiling, achieving performance nearly identical to the baseline without instrumentation.

Next, we enable multiple eBPF hooks—including kernel execution and memory transfer tracing—to assess profiling accuracy and overhead, as illustrated in Figure 5. eInfer delivers high accuracy while maintaining minimal overhead, closely matching the performance of CUPTI.

Our system is currently under development. Once fully implemented and integrated, we plan to design a range of case studies to evaluate its detailed performance breakdown. In the meantime, we measure the breakdown across host networking, CPU-GPU data transfer, and kernel execution time, as shown in Figure 6. With the full implementation, we will provide a more detailed breakdown of the performance components currently grouped under "others" in the figure.

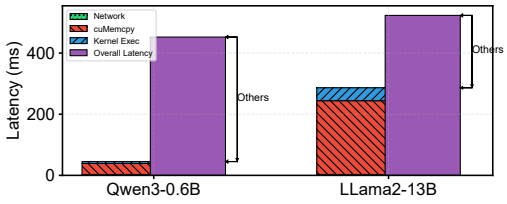


Figure 6: Performance Breakdown

4.1 Related Work

eBPF-Based System Instrumentation. eBPF enables low-overhead system introspection for CPU [10, 29, 34], memory [9, 31], and network [5, 12, 19] via tools like `bcc` [7] and `bpftrace` [18]. Microservice profilers such as ZeroTracer [32] and Deepflow [20] use eBPF for metric collection. However, eBPF's CPU-centric nature limits its applicability to GPU-intensive workloads.

GPU Tracing. Tools like CUPTI [16], FasterTransformer Profiler [17], and PyTorch Profiler [1] provide detailed GPU telemetry but are either hardware-specific or confined to user space [2, 3]. Recent systems such as `eGPU` [33] add eBPF-style probes on GPUs but introduce CPU-GPU synchronization overhead. Our evaluation confirms `eGPU`'s performance penalty under LLM inference, limiting its deployment for latency-sensitive applications.

Distributed Tracing. Frameworks like OpenTelemetry [28], Jaeger [24], and Dapper [25] trace request flows across services but lack low-level system or GPU integration. ML profilers [1, 2] offer execution insights but are typically single-node and user-space only. Sage [3] integrates system semantics but requires intrusive changes. Our work unifies eBPF with LLM-level tracing for fine-grained, cross-stack visibility in distributed inference.

5 CONCLUSION

We present eInfer, the first end-to-end, low-overhead system for transparent, fine-grained tracing of LLM inference across CPUs, GPUs, and interconnects using eBPF. Our key contributions include: (1) a novel approach to per-request correlation that spans across multiple components, devices, and distributed nodes without requiring application changes; (2) a unified, vendor-agnostic observability framework for both CPUs and accelerators that achieves near-parity with specialized tools like CUPTI; and (3) a scalable, runtime-adaptive tracing mechanism that maintains real-time visibility in complex, heterogeneous environments.

We are actively developing eInfer, with ongoing efforts focused on strengthening the robustness, scalability, and ease of deployment of the system. Once the implementation is complete, we will conduct comprehensive experimental evaluations to validate its accuracy, efficiency, and utility for LLM serving workloads.

REFERENCES

[1] [n. d.]. ([n. d.]). https://docs.pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[2] [n. d.]. ([n. d.]). https://www.tensorflow.org/tensorboard.

[3] [n. d.]. ([n. d.]). https://www.sage.com/en-us/sage-ai/.

[4] Amey Agrawal, Anmol Agarwal, Nitin Kedia, Jayashree Mohan, Souvik Kundu, Nipun Kwatra, Ramachandran Ramjee, and Alexey Tumanov. 2024. Etalon: Holistic Performance Evaluation Framework for LLM Inference Systems. *arXiv preprint arXiv:2407.07000* (2024).

[5] Ashima Chawla, Anne-Marie Bosneag, and Anestis Dalgkitsis. 2023. Graph-based interpretable anomaly detection framework for network slice management in beyond 5G networks. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–6.

[6] Yidong Chen, Chen Zhang, Rongchao Dong, Haoyuan Zhang, Yonghua Zhang, Zhonghua Lu, and Jidong Zhai. 2024. MixQ: Taming Dynamic Outliers in Mixed-Precision Quantization by Online Prediction. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[7] Brendan Gregg. 2019. *BPF performance tools*. Addison-Wesley Professional.

[8] Weifang Hu, Xuanhua Shi, Chang Wu, Yunkai Zhang, Xuan Peng, Jiaqi Zhai, Hai Jin, Yongluan Zhou, and Xuehai Qian. 2025. CFP: Low-overhead Profiling-based Intra-operator Parallelism Generation by Preserving Communication-Free Structures. *arXiv preprint arXiv:2504.00598* (2025).

[9] Kaiming Huang, Jack Sampson, Mathias Payer, Gang Tan, Zhiyun Qian, and Trent Jaeger. 2025. SoK: Challenges and Paths Toward Memory Safety for eBPF. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 810–828.

[10] Devki Nandan Jha, Graham Lenton, James Asker, David Blundell, and David Wallom. 2022. Holistic runtime performance and security-aware monitoring in public cloud environment. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 1052–1059.

[11] Andrew B Kahng, Robert R Nerem, Yusu Wang, and Chien-Yi Yang. 2024. NN-Steiner: A mixed neural-algorithmic approach for the rectilinear Steiner minimum tree problem. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 13022–13030.

[12] Abderaouf Khichane, Ilhem Fajjari, Nadjib Aitsaadi, and Mourad Gueroui. 2023. 5gc-observer: a non-intrusive observability framework for cloud native 5g system. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–10.

[13] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. 2024. Llm inference serving: Survey of recent advances and opportunities. *arXiv preprint arXiv:2407.12391* (2024).

[14] Yufei Li, Zexin Li, Wei Yang, and Cong Liu. 2023. RT-LM: Uncertainty-Aware Resource Management for Real-Time Inference of Language Models. In *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 158–171.

[15] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 2023. Specinfer: Accelerating generative large language model serving with tree-based speculative inference and verification. *arXiv preprint arXiv:2305.09781* (2023).

[16] NVDIA. [n. d.]. ([n. d.]). https://developer.nvidia.com/cupti.

[17] NVDIA. [n. d.]. ([n. d.]). https://github.com/NVIDIA/FasterTransformer.

[18] Mateusz Piotrowski. [n. d.]. Benchmarking Performance Overhead of DTrace on FreeBSD and eBPF on Linux. ([n. d.]).

[19] Cleiton Puttlitz, Ricardo Parizotto, and Alberto Schaeffer-Filho. 2024. P4NetIntel: End-to-End Network Telemetry with eBPF and XDP. In *2024 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 1–6.

[20] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, et al. 2023. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 420–437.

[21] Maohao Shen, Subhro Das, Kristjan Greenewald, Prasanna Sattigeri, Gregory Wornell, and Soumya Ghosh. 2024. Thermometer: towards universal calibration for large language models. In *Proceedings of the 41st International Conference on Machine Learning*. 44687–44711.

[22] Tianhui Shi, Jidong Zhai, Haojie Wang, Qiqian Chen, Mingshu Zhai, Zixu Hao, Haoyu Yang, and Wenguang Chen. 2023. Graphset: High performance graph mining through equivalent set transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[23] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[24] Yuri Shkuro. 2019. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd.

[25] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[26] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Esha Choukse, Haoran Qiu, Rodrigo Fonseca, Josep Torrellas, and Ricardo Bianchini. 2025. Tapas: Thermal-and power-aware scheduling for LLM inference in cloud platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1266–1281.

[27] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1348–1362.

[28] Xi-Zhen Wang and Chien-Chao Tseng. 2024. Building an Observable System Based on OpenTelemetry. In *2024 International Computer Symposium (ICS)*. IEEE, 76–81.

[29] Tianjun Weng, Wanqi Yang, Guangba Yu, Pengfei Chen, Jieqi Cui, and Chuanfu Zhang. 2021. Kmon: An in-kernel transparent monitoring system for microservice systems with ebpf. In *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*. IEEE, 25–30.

[30] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920* (2023).

[31] Bin Yang, Dian Shen, Junxue Zhang, Hanlin Yang, Lunqi Zhao, Beilun Wang, Guyue Liu, and Kai Chen. 2025. eNetSTL: Towards an In-kernel Library for High-Performance eBPF-based Network Functions. In *Proceedings of the Twentieth European Conference on Computer Systems*. 42–58.

[32] Wanqi Yang, Pengfei Chen, Kai Liu, and Huxing Zhang. 2025. ZeroTracer: In-band eBPF-based Trace Generator with Zero Instrumentation for Microservice Systems. *IEEE Transactions on Parallel and Distributed Systems* (2025).

[33] Yiwei Yang, Tong Yu, Yusheng Zheng, and Andrew Quinn. 2025. eGPU: Extending eBPF Programmability and Observability to GPUs. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems*. 73–79.

[34] Guangba Yu, Pengfei Chen, Pairui Li, Tianjun Weng, Haibing Zheng, Yuetang Deng, and Zibin Zheng. 2023. Logreducer: Identify and reduce log hotspots in kernel on the fly. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1763–1775.

[35] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with {eBPF}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1391–1407.

A HOOK POINTS

```
BPF_HASH(active_traces, u32, u64);  // pid -> trace_id mapping
BPF_PERF_OUTPUT(gpu_events);

int trace_ioctl(struct pt_regs *ctx, int fd, unsigned long cmd) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    u64 *trace_id = active_traces.lookup(&pid);
    // Check if trace ID exists and this is an NVIDIA ioctl
    if (!trace_id || (cmd & 0xFF00) != 0x4600)
        return 0;
    // Emit correlated GPU event with trace ID
    struct {
        u64 trace_id;
        u64 timestamp;
    } event = {
        *trace_id,
        bpf_ktime_get_ns()
    };
    gpu_events.perf_submit(ctx, &event, sizeof(event));
    return 0;
}
```

Figure 7: eBPF program for correlating GPU kernel submissions with request trace IDs

```
struct ckp_event {
    u64 timestamp;
    u64 op_id;      // Unique identifier (e.g., 0x1234)
    u32 op_type;    // KERNEL_LAUNCH, MEM_COPY, etc.
    char name[32];  // e.g., "GEMM_kernel_stream_3"
};

// CKP logs event to shared ring buffer
void log_kernel_launch(const char* kernel_name, int stream) {
    struct ckp_event evt = {
        .timestamp = rdtsc(),
        .op_id = generate_id(),
        .op_type = KERNEL_LAUNCH,
    };
    ring_buffer_write(&evt);  // Non-blocking write
}

// eBPF correlates with ioctl observations
int trace_gpu_ioctl(struct pt_regs *ctx) {
    // Match timing window: CKP event -> ioctl within 1ms
    struct ckp_event *ckp = find_recent_ckp_event();
    if (ckp && time_diff(ckp->timestamp, now) < 1000000) {
        // Correlated: Link GEMM launch to system-level GPU access
        emit_correlated_event(ckp->op_id, ckp->name);
    }
}
```

Figure 8: CKP-eBPF cooperation: User-space logs GPU operations, kernel-space correlates with system calls

Table 2: Hierarchical Instrumentation Points Across the Inference Pipeline

Stage	Hook Points	Captured Events
Request Entry	HTTP server `accept()`	Request arrival timestamp
	Python handler dispatch	Request ID, batch assignment
	Thread pool `work_queue`	CPU scheduling, queueing delay
CPU Processing	`sys_futex` (thread sync)	Thread coordination overhead
	`sys_mmap` (memory alloc)	Tensor allocation patterns
	`sched_switch`/`sched_wakeup`	CPU utilization, context switches
	`torch.nn.Module.forward()`	Layer execution boundaries
GPU Execution	`ioctl(/dev/nvidia*)`	Kernel submission, memory ops
	`cuLaunchKernel` uprobe	Kernel name, grid dimensions
	`gpu_mem_copy_start`/`done`	H2D/D2H transfer timing
	`dma_fence_signaled`	GPU completion events
	`nvml_device_get_utilization`	Real-time GPU metrics
Cross-Device	`nccl_send`/`recv`	Collective communication
	`ib_post_send` (RDMA)	InfiniBand operations
	`mlx5_cq_poll`	RDMA completion events
	`sys_sendmsg` (TCP/UDP)	Network fallback paths