MLCommons Chakra: Advancing Performance Benchmarking and Co-design using Standardized Execution Traces

Srinivas Sridharan [1] **Andy Balogh** [2] **Bradford M. Beckmann** [3] **Brian Coutinho** [1] **Louis Feng** [4] **Sheng Fu** [1] **Sanshan Gao** [1] **Mehryar Garakani** [5] **Taekyung Heo** [1] **David Kanter** [6] **Josh Ladd** [1] **Ziwei Li** [7] **Winston Liu** [2] **Changhai Man** [7] **Dan Mihailescu** [2] **Spandan More** [3] **Joongun Park** [7] **Ashwin Ramachandran** [4] **Vinay Ramakrishnaiah** [3] **Saeed Rashidi** [4] **Vijay Janapa Reddi** [8] **Puneet Sharma** [9] **Phio Tian** [1] **William Won** [3 7] **Hanjiang Wu** [7] **Huan Xu** [7] **Jinsun Yoo** [7] **Tushar Krishna** [7 10]

Abstract

The fast pace of artificial intelligence (AI) innovation demands an agile methodology for observation, reproduction and optimization of distributed machine learning (ML) workload behavior in production AI systems and enables efficient software-hardware (SW-HW) co-design for future systems. We present Chakra, an open and portable ecosystem for performance benchmarking and co-design. The core component of Chakra is an open and interoperable graph-based representation of distributed AI/ML workloads, called Chakra execution trace (ET). These ETs represent key operations, such as compute, memory, and communication, data and control dependencies, timing, and resource constraints. Additionally, Chakra includes a complementary set of tools and capabilities to enable the collection, analysis, generation, and adoption of Chakra ETs by a broad range of simulators, emulators, and replay tools. We present analysis of Chakra ETs collected on production AI clusters and demonstrate value via real-world case studies. Chakra has been adopted by MLCommons and has active contributions and engagement across the industry, including but not limited to NVIDIA, AMD, Meta, Keysight, HPE, and Scala, to name a few.

Chakra Codebase: https://github.com/mlcommons/chakra

1 Introduction

As artificial intelligence (AI) models continue to scale at an unprecedented rate in terms of size and capability, and large language models (LLMs) continue unlocking novel serving use cases, the design and deployment of the platforms on which AI training and inference is done of paramount importance. These AI data centers or AI supercomputers comprise of thousands to tens of thousands of customized neural processing units (NPUs) (e.g., NVIDIA Hopper/Blackwell, AMD Instinct™, Google Cloud TPU, Cerebras Andromeda) connected via high-speed scale-up/scale-out interconnects.

Unlocking performance from these platforms relies heavily on software-hardware (SW-HW) co-design across both the compute and communication stacks. Co-design is actually not a linear process, but rather a cyclic, iterative process.

Authors are listed in alphabetical order by last name, except for the corresponding authors. [1]NVIDIA [2]Keysight [3]AMD [4]Meta [5]Scala Computing [6]MLCommons [7]Georgia Institute of Technology [8]Harvard University [9]Hewlett Packard Enterprise [10]InfraVana.
Correspondence to:
- Srinivas Sridharan <srinivas@mlcommons.org>,
- Tushar Krishna <tushar@mlcommons.org>.

Proceedings of the 9th *MLSys Conference (Industry Track)*, Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

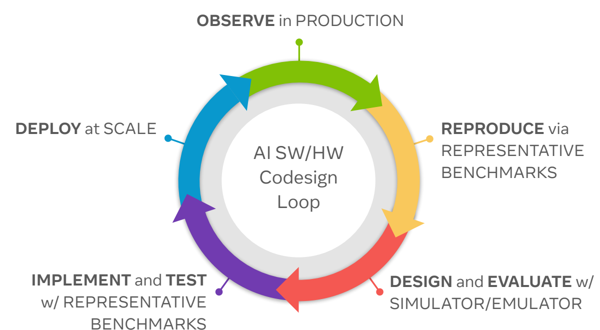


Figure 1. AI system SW-HW co-design flow.

Fig. 1 presents an overview of the co-design cycle that production platforms at hyperscalars go through from our experience. It involves observation and telemetry of the performance of current and emerging machine learning (ML) workloads to identify bottlenecks, reproducing these experiments via representative benchmarks, designing next-generation platforms to address (some of) these bottlenecks, evaluating these platforms via simulation (pre-silicon) and emulation (post-silicon), implementing and testing the platform (pre-deployment), and finally deploying at scale. This cycle continues.

This co-design cycle also highlights the suite of *frame-*

works and tools that are needed for enabling efficient co-design, spanning performance observation and analysis, reproducibility via replay, and simulators/emulators. All of these tools need to be driven by AI workloads representing current and future AI training/inference scenarios. Unfortunately, the ecosystem of these tools is extremely fragmented. Production workloads are developed and owned by AI labs and hyperscalars implemented in high-level frameworks (PyTorch/JAX), with internal observability tools, making it hard to reproduce behaviors in different environments. Simulators and emulators are a dime-a-dozen across NPU compute and networking vendors, of varying degrees of fidelity. Each of these have their own custom formats for describing workloads and the AI platform architecture. This fragmentation creates barriers to platform-agnostic analysis and co-design, and limits the opportunities for cross-platform optimizations. As a result, system-level optimizations are often limited to the boundaries of individual frameworks, restricting broader collaboration and innovation.

In addition, since open-source AI benchmarks, such as MLPerf, evolve slowly, there exist no efficient mechanism for workload sharing between the different players in the ecosystem. From our experience, hyperscalers and cloud service providers (CSPs) are hesitant in sharing proprietary model details, and often resort to sharing spreadsheets with representative model parameters under non-disclosure agreements with select partners. However, this still makes exact workload reproduction difficult. Many hardware vendors, startups and academic teams often derive parameters from public benchmarks, which can lead to over-optimization.

Furthermore, even with access to workloads and tooling, it is non-trivial to efficiently study optimizations on current systems and what-if scenarios for future systems. This is due to the extremely high-cost of running full-stack benchmarks as it requires (i) heavy cross-domain expertise[1] and (ii) access to expensive NPU clusters. Moreover, it is often hard to isolate specific HW/SW bottlenecks or compute versus memory, versus network behavior in end-to-end runs.

The fast pace of AI innovation demands an agile methodology to create workloads and rapidly iterate through the different stages of the co-design cycle. Identifying this need, Meta led a standardization effort in early 2023 (Sridharan et al., 2023) called "Chakra,"[2] in collaboration with Georgia Institute of Technology as an academic partner, to develop an open ecosystem of methodologies and tools to enable efficient performance benchmarking, optimization, and co-design for AI platforms. The central idea in Chakra is that of an *execution trace* (ET), a mechanism

[1]Companies have dedicated teams to optimize performance before MLPerf submissions for instance.

[2]Chakra means wheel in Sanskrit, and is inspired by the co-design cycle.

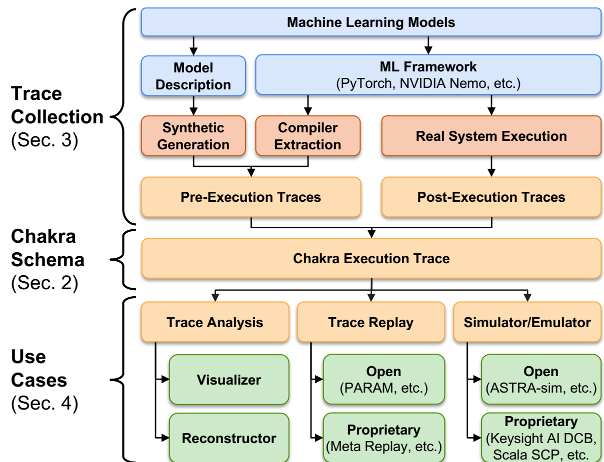


Figure 2. Chakra Infrastructure Overview.

to describe distributed AI workload performance behavior over an AI platform. Analogous to instruction and memory traces (Ranganathan & Victor), ETs record operator dimensions for compute and communication and their dependencies while avoiding disclosure of model or dataset details. Software organizations can share ETs of internal workloads with hardware vendors, who can in turn estimate and optimize performance using proprietary hardware models and simulators. The feedback loop helps software teams select compute and network configurations for training. The availability of ETs also enables academic researchers and startups to participate in co-design for production workloads. This methodology was successfully used between Meta and NVIDIA to identify challenges overlapping compute and comms in NVIDIA V100 systems (Rashidi et al., 2021a).

As several other companies started showing an interest in this vision, Chakra was formally adopted by MLCommons (MLCommons, 2023). Today, Chakra has evolved into a 40+ member working group actively engaged in schema standardization and ecosystem (methodology and tooling) development. The Chakra ecosystem is already being used by several companies, spanning hyperscalers and CSPs (Meta (Meta, 2023), Google), AI compute/full-stack vendors (NVIDIA, AMD, HPE, Intel, Marvell), system integrators (HPE), switch vendors (Juniper, HPE), simulation/emulation/testing tool suppliers (Keysight (Keysight Technologies, 2025), Scala Computing), and numerous academic groups and startups. Chakra trace collection is now officially supported as an option in both PyTorch and NVIDIA NeMo (NeMo). For simulation, Chakra is natively supported by the open-source ASTRA-sim (ASTRA-sim, 2025) distributed AI simulator, providing a reference implementation to the broader community, and by several proprietary simulators, including AMD internal simulation. Chakra has also enabled accessible benchmarking of distributed AI training on diverse platforms (Go et al., 2025).

Chakra Ecosystem. In this paper, we present the details of

the Chakra ecosystem. Fig. 2 illustrates its key components. The Chakra ET serve as the central element, facilitating the exchange of the model behavior needed for system optimization and co-design. Our contributions are as follows.

- **Schema Standardization.** We present the Chakra execution trace schema (Sec. 2) and surrounding workflow, shaped through collaboration with multiple organizations, that aim at standardization across platforms.
- **End-to-end workflow.** We present the co-design workflow enabled by Chakra, introducing trace collection methodologies from upstream ML frameworks (Sec. 3) and downstream use-cases spanning trace analysis, replay, and/or simulation/emulation (Sec. 4).
- **Case studies and artifacts.** We demonstrate our workflow via case studies spanning real trace analysis, replay benchmarks and simulation (Sec. 5). All these artifacts are being actively released as open-source via MLCommons to enable reproduction and adoption.

Sec. 7 contrasts Chakra with other related efforts.

2 THE CHAKRA SCHEMA

The Chakra schema provides a standardized representation of execution traces for AI workloads. The goal is to capture execution in a form that is portable across frameworks and extensible for diverse use cases, while remaining compact and easy to consume by diverse tools.

2.1 Design Requirements

ML tasks are naturally expressed as graphs, with frameworks such as TensorFlow and PyTorch constructing internal computational graphs (Minhas). In this work, we generalize this idea to *execution traces*, graphs that record not only model structure but also execution details such as memory accesses, compute operations, communication, and parallelization strategies. Although ETs are a powerful abstraction, their structure and metadata differ across frameworks, limiting reuse.

The Chakra schema was designed through discussions with both academic and industrial teams to meet several requirements. First, the schema must be minimal yet extensible: only essential fields should be mandatory, but attributes should allow customization without modifying the core definition. Second, the schema must be expressive for performance modeling: it should capture computation, memory, communication, and their dependencies. Third, the schema must be portable across levels of abstraction: traces should be usable for both real system replay and projection to future system configurations. Fourth, the schema should capture behavior at different stages of execution (Sec. 3).

Table 1. Chakra node schema.

Field Name	Data Type	Description
id	uint64	Unique node identifier
name	string	Human-readable name
type	enum(NodeType)	Node category (compute, memory, comms)
ctrl_deps	repeated uint64	Control dependencies
data_deps	repeated uint64	Data dependencies
start_time_micros	uint64	Optional start time
duration_micros	uint64	Optional duration
inputs/outputs	IOInfo	Values, shapes, types
attr	repeated AttributeProto	Extensible metadata

Table 2. Chakra communication schema.

Field Name	Data Type
type	enum(CommType): AllReduce, AllGather, ReduceScatter Broadcast, Point-to-Point, All2All, Barrier
group	uint64
tag	string
tensor_ids	repeated uint64

2.2 Schema Details

The schema represents execution as a directed acyclic graph (DAG) where nodes denote operations and edges encode data and control dependencies. Instead of fixing a large set of operators, Chakra defines a small set of node categories together with an attribute mechanism for extension. This ensures portability while allowing new operators or system-specific annotations to be added without breaking compatibility. Communication is modeled explicitly as a node type alongside computation and memory, enabling consistent representation of parallelization strategies and system-level effects. By unifying these elements, the schema supports performance analysis and SW-HW co-design without exposing proprietary model or dataset details.

Each node (Table 1) contains a unique identifier, a name, and a type that specifies whether it represents computation, memory, or communication. Control and data dependencies define the partial order, allowing the feeder to reconstruct execution. Optional timing fields capture start and duration hints. Inputs and outputs are described through `IOInfo`, which records identifiers, shapes, and data types of tensors. This allows each node to be linked consistently to the tensor schema without requiring a separate table in the main text.

Compute nodes describe compute operators either on host or device. They may expose different levels of details according to proprietary needs. The `dur` field is essential, as it reflects the computation duration. Fields such as `num_ops` and `tensor_size` can be used to specify high-level characteristics. Moreover, attributes like `kernel_name`, `launch_parameter`, and `kernel_args` can also be encoded to enable complete re-execution.

Communication nodes describe collective and point-to-point operations. The `type` field specifies the communication primitive. The `group` field specifies the set of ranks involved in the operation, and the optional `tag` helps distinguish concurrent operations. `tensor_ids` links the communication to tensor objects defined in the tensor schema.

Table 3. Tensor schema.

Field Name	Data Type
id	uint64
storage_id	uint64
storage_offset	uint64
shape	repeated int64
stride	repeated int64
dtype	fp16, bf16, int8, etc.
size_bytes	uint64

Table 4. Storage schema.

Field Name	Data Type
id	uint64
size_bytes	uint64
device	string

To support collective communication and parallelism across devices, Chakra uses a concept similar to *process groups* in PyTorch. In PyTorch, a process group defines a set of ranks that participate in a collective operation such as `all_reduce`, `all_gather`, or `reduce_scatter`. Chakra encodes the same idea by attaching a group identifier to communication nodes. This makes it possible to represent communication patterns over arbitrary subsets of NPUs. In practice, this allows traces to describe combinations of tensor, pipeline, data, and expert parallelism without altering the schema. Through process groups, Chakra can represent complex parallelization and communication strategies in a uniform way, while keeping the schema itself minimal and device traces independent.

Tensor descriptors capture the structure and placement of data. Each tensor has a unique identifier, shape, and data type, with the `size_bytes` field recording its storage footprint. The `storage_id` field records the physical memory on which this tensor is allocated with `storage_offset` to define the offset in that physical memory. Splitting tensor and its storage makes it possible to support tensor alias, for example, two tensors share the same storage but with different shapes. These descriptors allow compute and communication nodes to be linked consistently through tensor references. Tensor storage node represents a unique physical memory, it has a unique id, the memory size, and the device the storage resides. This design satisfies the requirements mentioned earlier Sec. 2.1. It is *minimal yet extensible*, containing only core fields while supporting rich metadata through attributes. It is *expressive*, capturing computation, memory, and collective or point-to-point communication. Finally, it is *portable*, accommodating traces collected at different abstraction levels, and consumable by diverse downstream tools.

Trace Storage. In the current implementation, each NPU maintains its own ET. For a workload with N NPUs, N distinct traces are generated. This design simplifies collection and replay since each NPU records only its local activity without referencing the execution of others. However, this per-device isolation reduces the opportunity for global scheduling, where work could be reassigned or balanced across NPUs. The Chakra schema does not prohibit such an approach, but its default mode of operation assumes per-device traces.

Trace Format. Chakra ETs started with the Google Protobuf format for size and efficiency considerations. Later, AMD introduced representing Chakra ETs in JSON format for better manageability (including human readability) and updated the ASTRA-sim simulator to support both formats of Chakra ETs to demonstrate the downstream tools can natively support both representations. Today, Chakra traces can be collected and used in both JSON and Protobuf.

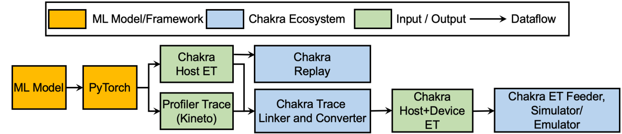


Figure 3. Chakra Trace Generation Flow from PyTorch

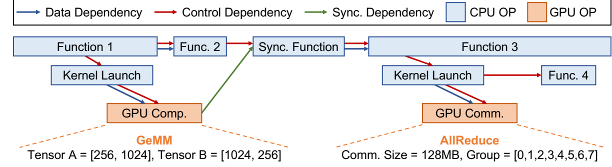


Figure 4. Converted Trace example.

3 Chakra Trace Collection

With the schema defined, the next logical step is to create traces. The Chakra ecosystem provides open-source tools for this, including a trace converter for post-processing native framework formats and a test case generator for synthetic trace creation. This tooling is complemented by native support in frameworks like PyTorch and NVIDIA NeMo, providing a direct path for trace generation. These tools and integrations support trace collection at two primary stages of the execution stack: pre-execution and post-execution, as shown in Fig. 2.

Pre-execution refers to a stage where an ML model has not yet been explicitly optimized or tied to specific system configurations, such as compute engine type, memory bandwidth, or network topology. Collecting a pre-execution trace allows for performance projection across various systems (current and future), as it is not tethered to a particular configuration. Conversely, the post-execution stage involves optimizing and executing the ML model on an actual system. These traces accurately represent and capture real-world phenomena. They are valuable for replaying of compute/-comms for tuning runtime libraries. However, these traces are tightly linked to the real system on which the model is run, with the parallelization strategy and several optimizations baked in. We discuss both of these in this section.

3.1 Post-Execution Traces

To collect post-execution traces, we leverage PyTorch's profiling stack with two complementary tools: execution graph observer (Feng) and Kineto (PyTorch, 2025). The execution graph observer records central processing unit (CPU) (i.e., host-side) execution, capturing the log-

ical sequence of operator invocations and control dependencies that describe how the CPU launches graphics processing unit (GPU) kernels. This forms the control-flow backbone of the workload. Kineto, in contrast, profiles device-side activity on GPUs (or NPUs), recording kernel launches, durations, and stream identifiers with microsecond precision. It provides the fine-grained timing information that the observer lacks. Both tools can be enabled through standard PyTorch application programming interfaces (APIs) or NVIDIA NeMo (NeMo) (see `NeMo/nemo/core/classes/modelPT.py`) without modifying user models. Furthermore, to capture modern LLM-era inference deployments, we extend Chakra's trace collection capabilities into serving frameworks like vLLM (vLLM) . Specifically, we integrate the collection of Chakra traces directly into vLLM's native profiling system (see `vllm/profiler/wrapper.py`).

Fig. 3 summarizes the Chakra trace generation flow. ❶ Running a PyTorch model produces a Chakra host ET (observer). Additionally, PyTorch can generate a profile trace (Kineto) that captures timing and device operator information. The Chakra host ET is sufficient for replay use cases since the trace is re-executed on a real system. ❷ The Chakra trace linker merges them into a unified dependency graph, resolving control, data, and synchronization edges. ❸ The Chakra trace converter validates the merged graph and emits a standardized Chakra ET following the Chakra schema. ❹ Downstream tools such as the ET feeder, simulators, and visualizers then consume the ET for analysis, and simulation. Each is described in detail in the subsequent sections.

3.1.1 *Chakra Trace Linker*

The *Chakra trace linker* merges host-side (CPU) and device-side (GPU or NPU) execution traces into a unified representation. Collected independently, these traces are otherwise disjoint, limiting holistic reasoning about control flow and performance. The linker resolves this gap by establishing explicit cross-domain dependencies and producing a single dependency graph in Chakra schema suitable for downstream tools. An example is shown in Fig. 4.

Why linking is necessary. Host traces collected through PyTorch's observer contain CPU activity and call stack information, while Kineto traces provide fine-grained timing information for kernels and operators. However, Kineto does not encode dependency information, and PyTorch observer lacks detailed device timing. Neither source alone is sufficient to construct a comprehensive dependency graph. To address this gap, Chakra combines both sources: CPU information from the host trace and timing from Kineto. We also contributed a patch to PyTorch so that observer and Kineto traces share common identifiers, allowing events to be matched across the two sources; this pull request has been merged upstream. Still, merging raw traces is not enough. In order to build a complete dependency graph suitable for analysis and simulation, we explicitly reconstruct multiple classes of dependencies—control, data, and synchronization—on top of the raw trace data. Concepts such as inter-stream and intra-stream dependencies, also discussed in systems like Lumos (Liang et al., 2025), are modeled in Chakra and integrated into the unified representation.

Control dependency. A control dependency indicates that the execution of one operation must follow another. For example, if function `A` calls function `B`, then `B` is control-dependent on `A`. Likewise, the return from `B` back to `A` also represents a dependency, since `A` cannot continue until `B` has completed. The linker reconstructs such causal ordering between host and device. This includes CPU function calls that launch kernels (CPU $\rightarrow$ GPU edges), returns or completion notifications from the device back to the host (GPU $\rightarrow$ CPU edges), and the ordering among host calls in the CPU call stack. These reconstructed links form the control-flow backbone of the unified trace and ensure that asynchronous operations are represented in the correct order.

Data dependency. Producer-consumer relationships are established by matching tensor identifiers and buffer handles across host and device. For example, when the CPU produces a tensor later consumed by a device kernel, or when a device kernel generates an output read by the host, the linker inserts explicit data edges. Similarly, two GPU kernels have producer-consumer relationship such as compute operation followed by communication operation (inter-stream dependency), compute operation followed by another compute operation (intra-stream dependency), or hierarchical collective communication (inter-stream dependency) linker connects those with data edges. This guarantees that data movement and its dependent computation are represented consistently.

Synchronization dependency. The linker also inserts edges that capture ordering imposed by synchronization operations. This includes global synchronizations such as `cudaDeviceSynchronize()`, stream-specific calls like `cudaStreamSynchronize()`, and event-based coordination through `cudaEventRecord()` and `cudaStreamWaitEvent()`. Inspired by Holistic Trace Analysis (FacebookResearch), these synchronization edges are treated as additional dependency constraints that extend the unified graph. By encoding them, the linker enables critical-path analysis and ensures that both explicit and implicit synchronization points are visible to downstream tools.

3.1.2 *Chakra Trace Converter*

The Converter operates after the linker and has two goals: (1) verify the dependencies produced by linking, and (2) emit a standardized Chakra ET graph.

Dependency verification. The Converter checks the linked

graph for structural soundness and removes artifacts that hinder analysis. It enforces acyclicity via standard topological validation, prunes false or redundant edges (e.g., edges contradicted by per-stream order or duplicating implied relations), reconciles inter- and intra-stream constraints into a consistent ordering, and validates process group and domain consistency for communication nodes. All surviving edges are normalized into a single edge set with a `dep_type` label (control/data/sync) and de-duplicated for determinism.

Graph emission. Verified events are serialized into the Chakra schema as typed nodes (`COMP`, `MEM_LOAD/MEM_STORE`, `COMM_SEND/RECV/COLL`) with optional timing hints (`start_time_micros`, `duration_micros`) and extensible attributes (tensor sizes, ranks, group identifiers, tags, input-output (IO) metadata). Edges are emitted as a cycle-free, canonical adjacency with stable ordering. The result is a single, standardized Chakra DAG.

3.2 Pre-Execution Traces

Existing work has explored synthetic or compiler approaches for pre-execution trace generation. **STAGE** (Man et al., 2026) constructs symbolic tensor graphs enabling scalable synthesis of LLM benchmarks without relying on a specific runtime environment. **SimAI** (Wang et al., 2025) has a custom domain specific language (DSL) to describe LLMs and create a workload graph (that we have successfully ported to the Chakra schema) with a focus on detailed kernel-level modeling of computation and communication, enabling more accurate runtime-specific performance analysis. **AMD** leverages an in-house analytical modeling framework to evaluate the performance of end-to-end AI application execution and generates pre-execution Chakra graphs. **Flint** (Yoo et al., 2026a) captures PyTorch compiler's Intermediate Representations (`torch.fx`) and converts them into Chakra graphs. **LayerDAG** (Li et al., 2024) leverages generative AI models to synthesize execution traces in order to obfuscate intellectual property while retaining representative performance characteristics.

These approaches highlight complementary motivations for different trace synthesizers: STAGE emphasizes scalability and coverage of large (and hypothetical) models, SimAI and AMD tool emphasize accuracy for specific platforms, Flint allows users to easily obtain high fidelity graphs from model source code as-is. LayerDAG emphasizes protecting intellectual property from trace release. Chakra's schema accommodates for all these pre-execution traces for design space exploration.

4 Chakra Downstream Use Cases

Inspired by the co-design stages shown earlier in Fig. 1, the Chakra ecosystem provides tooling and harnesses for three broad tasks: analysis, replay and simulation/emulation that are required at different times within the development cycle of AI platforms. The open schema enables interoperability across different stages and diverse open/proprietary tools.

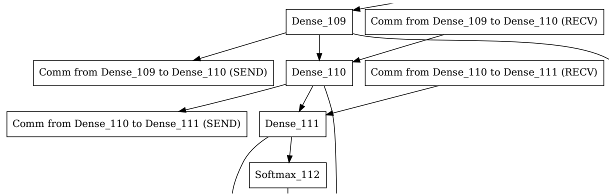


Figure 5. Chakra ET visualization example.

4.1 Trace Analysis

Chakra offers a range of open-source tools to help users visualize, analyze, and consume execution traces. We describe these here briefly. All these tools are available open-source at the MLCommons Chakra GitHub repository and are under active development.

Trace Visualizer. The trace visualizer is designed to visualize execution traces in the Chakra schema—both the dependencies and the timeline. An example of a visualization from the execution trace visualizer is presented in Fig. 5. The visualizer is a helpful tool for researchers to understand the structure of execution traces, or debugging. By default, the visualizer encodes the names of tasks and dependencies between tasks. Users can easily modify the visualizer to encode additional metadata such as compute time and communication size.

Trace Reconstructor. A trace reconstruction tool consumes a Chakra ET graph and executes a policy-agnostic topological schedule (Kahn-style ready queue), which can be used for validation, benchmarking or visualization.

Dependency-Aware ET Feeder The feeder ingests execution traces as a dependency graph and streams nodes to a simulator while strictly preserving the partial order defined by control and data edges. To bound memory usage, nodes are read in windows rather than loading the entire trace. When a node refers to a parent that has not yet appeared, it is placed into an unresolved set until the parent arrives. The feeder then elastically extends the window to resolve cross-boundary dependencies, ensuring correctness without preloading the full trace. Each node maintains a count of unresolved predecessors. A node is ready once this count reaches zero and is inserted into a ready queue. The queue itself is policy-driven: nodes may be prioritized by measured start time, communication priority, or simply issued in first-in-first-out (FIFO) order. Crucially, policies only arbitrate among ready nodes and therefore cannot violate dependency invariants. When a node completes, the feeder decrements the predecessor counts of its children, potentially unlocking new ready nodes. This process ensures that the emission order respects the original dependency structure.

With this design, correctness is guaranteed by construction, while scheduling policy remains pluggable. The approach yields deterministic emission orders under a fixed policy, scales linearly with the size of the trace, and keeps memory usage proportional to the window size rather than the entire input. As a result, the feeder can support a variety of backends—compute simulators while preserving the causal structure of the original execution trace.

4.2 Chakra Trace Replay on Current Systems

Benchmarking for ML systems has traditionally relied on carefully chosen applications that are adapted or simplified to expose specific behaviors. This process is time-consuming and difficult to maintain as workloads evolve.

Chakra replay takes an alternative path by relying on AI framework backend, e.g. PyTorch Aten and c10d backends, to re-execute compute and communication operations in ETs at scale on real systems. By using traces from production, collected with low-overheads, we can quickly generate portable benchmarks that can be used early stage platform evaluation, subtrace replay, and scaled-down performance testing. This approach achieves high fidelity with production workloads as demonstrated for communication operations (Research) as well as full workloads (Liang et al., 2023).

MLCommons is considering defining networking benchmarks using this methodology as well as improving existing MLPerf Storage benchmarks (MLPerf) to use Chakra replay.

4.2.1 Chakra Replay Benefits

Chakra replay offers several notable advantages in comparison with re-running the original workload.

Data Privacy and Accessibility. The replay process substitutes randomized input data wherever feasible to mitigate data-privacy concerns inherent in production environments. This design choice allows researchers and performance engineers to analyze workload behavior without requiring access to sensitive user data, which is often restricted.

Accelerated Debugging and Analysis. A Chakra trace encapsulates only the operations executed on the target devices. Consequently, the replay tool bypasses the model initialization and compilation phases—such as the potentially time-consuming PyTorch model compilation step—thereby substantially reducing turnaround time during debugging and performance analysis.

Fine-Grained Replay Control. The tool allows users to replay a selected subsequence of operators rather than the full trace. This capability facilitates targeted examination of specific computational regions for debugging or performance optimization.

Device Agnosticism. Because Chakra traces record operators at the ML framework level, they are typically device-independent. As long as the recorded operators are supported on the target hardware (for example, communication collectives), the replay can be executed on devices different from those used during the original trace collection.

4.2.2 Chakra Replay Workflow

The replay tool operates through the following stages.

Process Initialization. A dedicated process is instantiated for each rank and corresponding device. Each process loads the Chakra trace associated with its rank.

Trace Parsing. The trace is parsed to identify the set of nodes to be executed according to the selected replay configuration—compute-only, communication-only, or full replay—and to determine whether to execute all operators or a user-specified subset.

Operator Initialization. The relevant operators are instantiated and initialized through the corresponding machine-learning framework interface.

Tensor Allocation. Input and output tensors are analyzed, and memory is provisioned according to the chosen tensor allocation strategy:

Pre-allocation mode allocates all input tensors before execution, trading increased memory consumption for lower allocation overhead. Lazy allocation mode allocates tensors on demand and releases them when they leave scope, improving memory efficiency at the expense of additional allocation overhead.

Execution and Profiling. The operators are executed in the original recorded order. When profiling is enabled, the tool produces a detailed performance report, including kernel-level timing and execution statistics.

4.2.3 Collectives Accuracy Comparison

One of the challenges in AI model training is maintaining consistent model convergence across various device accelerators. The new accuracy checker feature in Chakra replay addresses this by comparing the outputs of collective reductions. This is achieved by replaying the input collective tensors on different accelerators.

The accuracy comparison capability within the Chakra replayer facilitates the following:

- Validation of the relative difference of collective reduction outputs between diverse accelerator hardware and reduction algorithms.
- Comparison of the precision loss with different datatypes both on the same accelerator type or across different accelerators types.
- Debugging model collectives by replaying the model's

input tensors to analyze the reduction behavior on the accelerator.

Specific performance results are not presented here due to proprietiary nature of analysis.

4.3 Performance Projection for Future Systems

4.3.1 What-if Analysis with Simulators

Traditional compute or network simulations often rely on synthetic or overly simplistic workload models that fail to capture the nuanced complexity and dynamism of AI applications. As AI workloads increasingly dominate datacenter traffic, network research, in particular, must reflect their unique characteristics, including highly variable traffic patterns, burstiness, latency sensitivity, and high-bandwidth demands. To this end, Chakra traces serve as workload specifications for forward-looking system studies. Here, the trace supplies operator types, tensor shapes, and dependency structures, while simulators or performance models can provide execution time breakdowns. In particular, given the coarse-grained compute adn communication operators in the Chakra schema, simulators can also be used to study software optimizations for compute and collective algorithms, which is not feasible when running over a fixed vendor software stack (e.g., CUDA and NCCL from NVIDIA or RCCL™ from AMD).

ASTRA-sim (ASTRA-sim, 2025) was one of the first open-source simulators to adopt Chakra, leveraging Chakra's ET feeder (Sec. 4.1) to replace its custom workload format. This has enabled several co-design studies studying novel platforms—spanning new fabric topologies (Won et al., 2024b), wafer-scale systems (Rashidi et al., 2025), and topology-aware collective synthesis (Won et al., 2024a). Today, Chakra trace support has also been enabled within proprietary simulators, e.g., by Scala Computing (Scala Computing, 2025).

4.3.2 Hardware-in-the-Loop Validation with Emulators

A key industry trend for accelerating multi-year AI supercomputer development is to *shift left*, moving validation from full-scale clusters to earlier, component-level validation (Wareing & Graf, 2025; Bergeron & Kumar, 2025). This hardware-in-the-loop (HIL) validation relies on emulators driven by high-fidelity workloads. Here, the Chakra ET serves as the portable workload specification. An emulator replays the trace to generate at-scale network dynamics, directing this stimulus at a physical device under test (DUT) like a network interface controller (NIC) or switch. This process allows engineers to verify performance against simulation predictions and uncover implementation bugs that real-world traffic patterns expose. The viability of this trace-driven approach is demonstrated by its native adoption in open-source tools like Genie (Yoo et al., 2026b) as well as commercial system emulators like Keysight AI Data Center Builder (Keysight Technologies, 2025), which now support the Chakra format for workload import and replay.

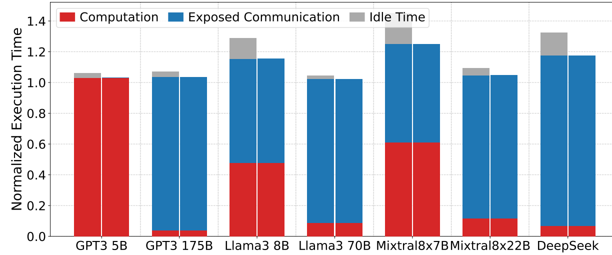


Figure 6. Normalized execution time breakdown across workloads for traces collected on the system mentioned in Sec. 5. For each workload, we show measured performance from Kineto (left) and the performance via trace reconstruction through Chakra (right).

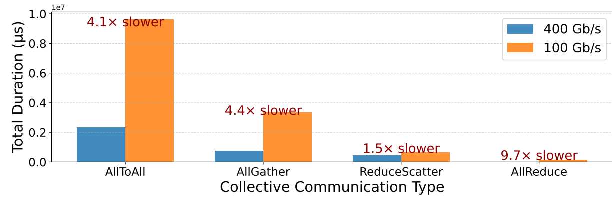


Figure 7. Total collective communication runtime comparison at 400 Gb/s and 100 Gb/s InfiniBand. Measured on training Mixtral-8×22B with 32 GPUs (four HGX-8×H200 nodes, TP/SP=4, EP=8) and the global batch size of 32.

5 Evaluation

We present a suite of case studies using a mix of open and commercial tools that support Chakra today to demonstrate the value of the ecosystem to the community.

5.1 Chakra Trace Analysis

Evaluation System. We collect traces from the Georgia Tech AI Makerspace (Georgia Institute of Technology, 2026), a supercomputer hub providing a diverse set of largescale compute resources to students and researchers. Here we use 128 NVIDIA H100/200 GPUs interconnected with Infiniband HDR100 (100 Gb/s), dual 32-core Intel Sapphire Rapids CPUs, and DDR5 DRAM. We also collect traces from Hewlett Packard Enterprise, across 32 NVIDIA H200 GPUs interconnected with Infiniband HDR400, dual 32-core Intel Xeon 8562Y CPUs, and DDR5 DRAM. Our software environment includes PyTorch 2.5, NVIDIA NeMo 24.07, and Megatron 0.10.0. We track control and data flow by analyzing the profiling results in both trace view and graph execution format. The details of the traces collected for this study are presented in Table 5. We have made these traces publicly available online.[3]

[3]https://github.com/mlcommons/chakra/wiki/Chakra-Trace-Library

Table 5. Counts of key operations per GPU for a single epoch.

Model	GPUs	Parallelization	Computation				Communication				
			GeMM	Attn	ElemWise	Others	P2P	AllReduce	All2All	AllGather	ReduceScatter
GPT3 5B (Brown et al., 2020)	8	TP=8, w/ SP	37,248	6,144	24,832	165,207	0	514	0	18,816	12,416
	8	PP=8	4,608	768	3,072	21,475	136	2	0	0	0
	8	FSDP=8	4,656	768	3,104	9,400	0	17	0	784	400
GPT3 175B (Brown et al., 2020)	32	TP=32 w/ SP	36,960	6,144	24,640	88,573	0	130	0	18,528	12,320
	64	TP=4, DP=2, PP=8, w/SP	9,216	1,536	3,072	26,615	67	67	0	4,683	3,078
Llama3 70B (Grattafiori et al., 2024)	8	TP=4, PP=2	24,576	4,096	12,288	46,080	130	2	0	12,416	8,320
	8	TP=8	49,536	8,192	24,832	85,634	0	16,897	0	0	0
	16	TP=4, PP=2, DP=2	12,288	2,048	6,144	23,054	65	4,161	0	16	16
Mixtral 8x7B (Jiang et al., 2024)	8	TP=2, EP=4	5912	1024	7751	16657	0	291	1024	1322	1122
	32	EP=8, PP=4	1,920	256	512	11,028	16	2	512	9	9
	128	TP=4, EP=8, PP=4	1,920	256	512	10,801	16	131	512	659	531
Mixtral 8x22B(Jiang et al., 2024)	32	EP=8, TP=4	3,596	896	6,333	13,277	0	243	896	1,134	905
DeepSeek-MoE (Dai et al., 2024)	8	EP=8	27,456	896	6,784	46,938	0	18	1,728	23	23
DLRM (Naumov et al., 2019)	8	MP=8	77	0	152	878	5	9	10	0	0
Resnet50 (He et al., 2015)	2	MP=2	33	0	347	3263	0	15	0	0	0

❶ **Runtime Analysis.** Fig. 6 shows each workload evaluated under comparing the Kineto and Chakra traces. Computation (red) and exposed communication (blue) in Kineto and Chakra align closely. However, for co-design analysis, Chakra excludes the idle time (gray) that occurs between different GPU and CPU kernels.

Fig. 7 presents the Chakra trace analysis results for the Mixtral-8×22B model executed with identical tensor and expert parallelism configurations across InfiniBand networks of different bandwidths. Because the scale-up interconnect is confined to eight GPUs per node, it is primarily utilized for tensor parallelism (TP), particularly for non-mixture-of-experts (MoE) components, while most expert related communications occurs over the slower scale-out InfiniBand network. Using Chakra for analyzing total duration for different collective communications, we observe that a 4.0× slower InfiniBand bandwidth leads to an approximately 4.1× and 4.4× slowdown in All-to-All and All-Gather, respectively. Also, higher-bandwidth InfiniBand also exhibits lower latency, so the effective slowdown is less than the theoretical 4× ratio and this effect becomes more prominent in the All-Reduce kernel, which has substantially smaller communication volume.

Additionally, by encoding memory utilization collected from Kineto for compute nodes, Chakra tracks the total utilization at different timestamps throughout the training epochs. Fig. 8 illustrates the memory usage contributed by various compute nodes during the execution.

❷ **Kernel Analysis.** Breaking down end-to-end computation and communication into individual kernel nodes, Chakra extracts precise operation counts and timings across diverse models, parallelization strategies, and configurations, and encodes them in a consistent graph representation. Table 5 summarizes the raw operation counts across various models, while Fig. 9(a) and Fig. 9(b) visualize the cumulative distribution function (CDF) of compute durations and data dependencies obtained from the exemplified Mixtral-8x22B trace.

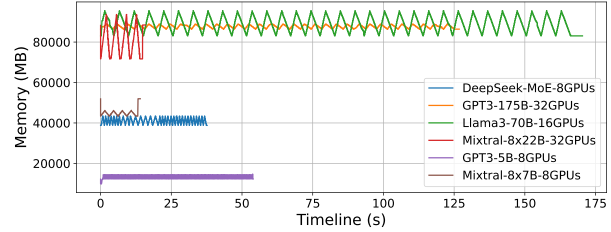
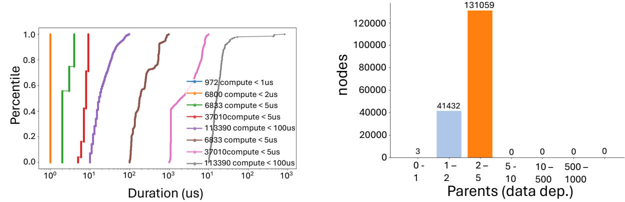


Figure 8. GPU memory utilization for different LLM models during one training step. Traces are aligned relative to the start of each epoch. Each model and its corresponding parallelization match the first entry (row) in Table 5 using the same number of GPUs.



(a) Cumulative distribution function of node durations.

(b) Distribution of data dependencies.

Figure 9. Compute characteristics of the Mixtral-8x22-Chakra trace. (a) Most compute kernels complete within 2–10^2 µs. (b) The majority of nodes have 10–500 parent data dependencies.

5.2 Trace Replay Case Studies

Replaying Chakra ETs on real systems allows reproducing the exact workload behavior either fully (replay both compute and comms operations) or partial replay (replay selective operations). The latter enables users isolate specific system components from overall workload execution to identify performance optimization opportunities (e.g. focus on networking improvements).

We present a case study for replaying communication operations for a Megatron-LM 43B GPT model (48 transformer layers) on four H100 nodes (32 ranks) with pipeline parallelism (PP) factor of four, TP factor of four, and data parallelism (DP) factor of two. Table 6 shows the NCCL kernel bus bandwidth report from Chakra communication

Table 6. NCCL kernel bandwidth report from Chakra replay for top 10 NCCL kernels by message size, across different process group sizes.

Kernel	Size	Rks	Dur (ms)	BW Base (GB/s)	BW Ratio
ReduceScatter f32	9.0G	2	15.790	243.6	1.261
ReduceScatter f32	9.0G	2	57.378	157.7	1.106
AllGather	4.5G	2	7.848	289.2	1.065
AllGather	4.5G	2	7.491	312.8	1.034
ReduceScatter f32	404M	2	22.428	153.9	1.214
ReduceScatter f32	320.1M	2	0.778	174.6	1.236
ReduceScatter f32	256M	2	3.887	177.5	1.013
AllGather	202M	2	1.264	209.0	0.784
ReduceScatter f32	192.1M	2	0.494	170.3	1.197
AllGather	160.0M	2	0.395	164.6	1.291

replay. The NCCL kernel bandwidth is close but typically faster than the original kernel in the baseline run due to lack of memory contention on overlapping with compute.

The replay methodology helped a vendor identify a set of communication operations (AllGather and ReduceScatter) across microbatches that could be run in parallel leading to a 2× improvement in communication collective performance

5.3 Hardware-in-the-Loop Emulation Case Study

We present a case study demonstrating the HIL validation methodology described in Sec. 4.3.2. The experiment investigates the performance of a physical Ethernet fabric under a realistic MoE workload to uncover emergent, system-level bottlenecks related to congestion control.

Emulation System. The physical system under test (SUT) consisted of a four 12.8T switch fabric arranged in a 1:1 Clos topology at 400 Gbps port speed. The fabric was configured with standard data center quantized congestion notification (DCQCN) for congestion control. Realistic AI traffic was generated using Keysight AI DCB, a system emulator capable of replaying Chakra ET to generate high-fidelity remote direct memory access (RDMA) traffic (Bortok, 2025). This allows for injecting precise, workload-specific communication patterns into the physical SUT.

Target Workload. The experiment utilized a synthetic Chakra ET designed to model the communication patterns characteristic of a modern MoE training iteration. Unlike simpler models that rely primarily on a single type of collective, MoE workloads are defined by the frequent interleaving of both All-Reduce and All-to-All collective operations. These two collectives represent opposite extremes of communication patterns: All-Reduce operations typically involve a few high-bandwidth flows, while All-to-All operations create a mesh of many low-bandwidth flows. This mixed-collective pattern was injected into the SUT to analyze its performance under realistic contention.

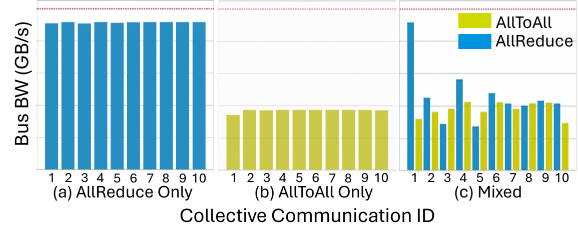


Figure 10. Bus bandwidth per iteration when (a) All-Reduce (b) All-to-All (c) mixing All-to-All and All-Reduce in one time span.

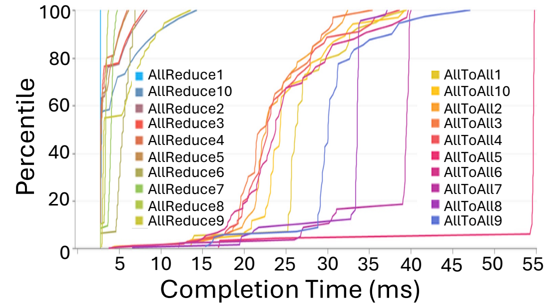


Figure 11. Mixing collectives results of CDF.

Result. The experiment revealed a significant performance anomaly when interleaving All-Reduce and All-to-All collectives. While both collectives show stable performance in isolation (Fig. 10(a) and (b)), the mixed workload in Fig. 10(c) demonstrates a mutually detrimental interaction. The most visible impact is on the All-Reduce operations. After being interleaved with an All-to-All collective, their performance becomes highly variable and fails to consistently achieve the near line-rate bandwidth seen in isolation. Concurrently, while the aggregate throughput of All-to-All appears stable, its constituent flows are negatively impacted. Upon investigating, we found that the high-rate All-Reduce flows trigger DCQCN's congestion control mechanism, which in turn disproportionately throttles the many small flows of the All-to-All, creating stragglers. Fig. 11 discusses the CDF for concurrent execution of two collective communicaitons (All-to-All and All-Reduce) extended from the isolated. We observe that the mixing incurs a long-tail *flow* completion time distribution for All-to-All creating stragglers that increase overall *job* completion time.

5.4 Trace Simulation Case Studies

Next, we demonstrate the inter-operatability Chakra by running it through two distinct simulators. The first is the open-source ASTRA-sim (ASTRA-sim, 2025) and the second is the commercial Scala Compute Platform (SCP).[4]

5.4.1 Different Network Topology and BW using AstraSim

Setup. We assume a system with eight H100, and we vary the network topology between switch, ring, and fully-

[4]SCP (Scala Computing, 2025) is a simulator built over ns-3 to perform large-scale simulations in AWS Cloud.

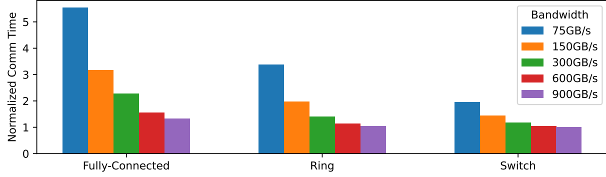


Figure 12. Communication time for different network topology and bandwidth with Mixtral 8x7B target.

connected. Additionally, we test bandwidths ranging from 75 GB/s to 900 GB/s. The Mixtral 8×7B model serves as the workload for this evaluation.

Results. Fig. 12 illustrates the normalized communication time of workloads while varying the network bandwidth. We observe the following: (1) Even with the same end-link bandwidth, different network topologies exhibit distinct communication times. In general, the switch topology achieves the best performance, followed by ring, and finally fully-connected. We suspect this is because the switch topology enables better link utilization, whereas other topologies contain links that are not actively used, especially in the fully-connected case. (2) As the bandwidth increases, the communication time eventually converges and stops improving, since latency becomes the dominant factor compared to bandwidth. This observation suggests that to further accelerate communication, one may need to increase the chunk size to mitigate the impact of latency.

5.4.2 Network Transport Analysis using SCP

Setup. We simulated a 256 A100 GPU system with fully-provisioned 51.2 Tb/s ethernet switches in a two-tier Clos topology with 800 Gb/s inter-switch links and 400 Gb/s NICs, running RDMA over converged Ethernet (RoCE) v2. We simulated a Llama-2 trace (Man, 2024), with a 64 DP × 2 TP × 2 PP × 1 SP parallelism.

Results. The distribution of NICs by transmit utilization is shown in the Fig. 13(a) heatmap. Notably, after 300 ms, a large fraction of NICs are no longer transmitting at high rates. This arises from the synchronous nature of LLM workloads: prolonged intervals of low network utilization reflect GPUs spending substantial time on computation rather than communication (due to the nature of more DP than TP and PP in the experiment setup) or waiting to receive data from other GPUs. Considering the receive-side NICs in Fig. 13(b), we observe oscillatory behavior driven by the workload structure. Even with a fully provisioned topology, the average link utilization remains well below the 400 Gb/s line rate, with extended periods of underutilization while GPUs are computing or waiting on dependencies. In summary, Chakra provides an effective way to capture dependencies within and across GPUs, enabling fine-grained simulation of LLM workloads and enable studying the effect of network transport on end-to-end performance.

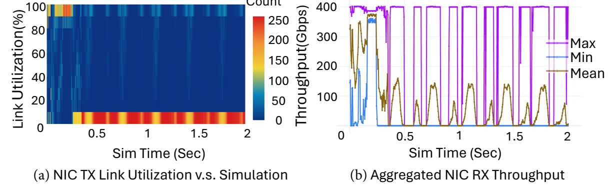


Figure 13. Results of RoCE v2 NIC transmit receive.

Table 7. Comparison of KV-cache offloading and baseline (no offloading) operations in Chakra collected traces for Llama3-8B.

Operation	Baseline Count	Baseline GPU Time (ms)	Offloading Count	Offloading GPU Time (ms)
Memcpy HtoD	2,334	2.088	4,857	11.400
Memcpy DtoH	387	0.895	5,958	216.484
`start_load_kv`	N/A	N/A	784	6.337
`start_store_kv`	N/A	N/A	776	215.068

5.5 Inference Trace Analysis

As introduced in Sec. 3.1, Chakra's native trace collection capabilities were originally designed for training workloads in frameworks such as PyTorch and NVIDIA NeMo, and are now being extended to inference engines like vLLM to capture emerging and dynamic inference mechanisms not present in training. Here, we analyze traces collected through our ongoing vLLM integration across three key inference mechanisms. All experiments are conducted on vllm v1 (vLLM, 2026) with supportance of prefill/decode disggregation. The same GPU/CPU platform is used as in Fig. 5.1, at a smaller scale than our training studies due to the nature of inference.

5.5.1 MoE Token Routing

Unlike training frameworks that pad or drop tokens to maintain balanced All-to-All communication in NeMo, inference workloads preserves every token and creates a dynamic and load-imbalanced workload pattern that will be assigned across different experts being held on different GPUs. In Chakra, we embed the per-expert bin counts (e.g. [1, 2, 5, 0, 0, 0, 0, 4] for eight experts) to the MoE routing nodes. Fig. 14 shows the profiled results of MoE token routing among two GPUs, so each GPU rank will hold four experts and receive an imbalanced amount of tokens.

5.5.2 Memory Offloading between CPU and GPU

To demonstrate Chakra's ability to analyze memory offloading, we collect traces by forcing key-value (KV) cache offloading under limited GPU memory. We then isolate trace nodes captured from the recorded function `start_load_kv` in vLLM's KV connector. Table 7 quantifies the impact of offloading by tracking KV store/load events and the resulting extra host (CPU)–device (GPU) memory interactions, highlighting Chakra's ability to expose system-level bottlenecks in memory-bound inference settings.

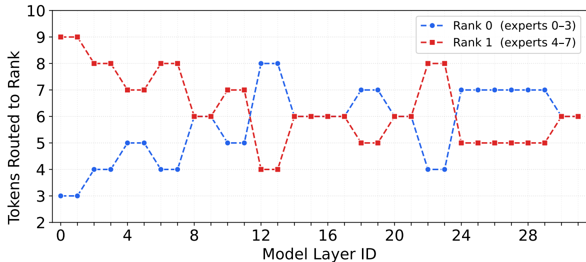


Figure 14. Distribution of token routing among two expert parallel rank for each model layer. The input has six tokens and the model used is Mixtral 8x7B with 32 layers.

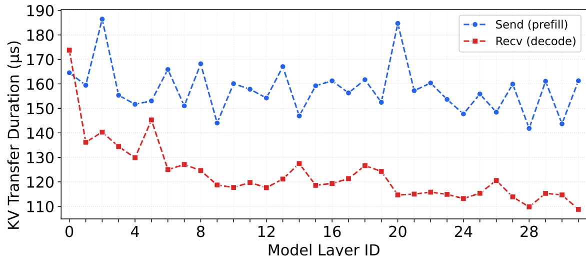


Figure 15. Runtime breakdown of the KV cache transfer for inferencing Llama3-8B between one prefill and decode GPU. The captured trace denotes the per-layer (32 layers for Llama3-8B) send and receive latency between two GPUs.

5.5.3 KV-Cache Transfer

In inference, when disaggregating prefill and decode stages on different GPUs (Patel et al., 2024; Zhong et al., 2024; Bambhaniya et al., 2026), it introduces unique point-to-point communication requirements. To capture this, we profiled the message sizes of the point-to-point messages responsible for transferring the KV cache between dedicated prefill and decode GPUs as shown in Fig. 15.

6 Lessons Learned and Extensions

We discuss lessons learned while working with Chakra. We then point out how our lessons evolved into current and future extensions.

6.1 Lessons Learned

- A graph-based ET effectively bridges the gap between high-level frameworks (e.g. PyTorch), LLM frameworks (e.g. NeMo), and downstream replay/simulation tools; provide a common abstraction from multiple industry participants to exchange critical bottlenecks without revealing sensitive intellectual property (IP).
- By abstracting execution into compute, memory, and communication nodes with explicit dependencies, we can enable portable and reproducible benchmarking without disclosing model weights or datasets. This has allowed Meta/NVIDIA to identify bottlenecks in overlapping compute/comms that were invisible in isolated benchmarks (Rashidi et al., 2021b).
- While the original Chakra capture mechanism was mainly designed for eager mode operator execution, we continue to expand the trace to include information like synchronization dependencies, KV transfers and data loader nodes for MLPerf Storage benchmarks. This enables the trace to provide a higher fidelity represntation of the actual system behavior.
- Our experience reaffirms a cyclic process from productive SW/HW codesign: observe (in production) → reproduce (via replay) → design/evaluate new solutions (via simulators) → deploy (in production). To summarize, Chakra acts as the standardized mechanism for representing key performance aspects of a workload through every stage of the co-design cycle.

6.2 Ongoing Extensions

6.2.1 Handling Large-Scale Traces

The Chakra framework prioritizes completeness and fidelity in trace representation, which is essential for accurate performance modeling and replay. For very large workloads, such as billion-parameter language models or training runs with millions of operations, this design choice results in execution traces that can span multiple gigabytes. While these trace sizes reflect the true complexity of modern ML workloads, they also present practical challenges for storage, exchange, and initial loading. This approach represents an intentional trade-off: preserving full operational detail enables high-fidelity simulation and faithful bottleneck analysis, which would be compromised by lossy compression or aggressive summarization. To improve usability at scale without sacrificing fidelity, future enhancements could include optional lossless compression for storage and transfer, alongside hierarchical indexing to enable partial loading and selective replay. Such optimizations would enhance the scalability of the framework while maintaining its core accuracy guarantees for large-scale analysis.

6.2.2 Infrastructure Abstraction

While Chakra provides a common representation for workload execution traces, infrastructure descriptions still rely on tool-specific formats that limit portability and cross-platform comparison. Just as Chakra enables workload exchange without exposing proprietary model details, a standardized infrastructure abstraction would allow hardware configurations to be shared and evaluated systematically across teams and organizations. Graph-based infrastructure representations that capture compute nodes, memory hierarchies, and interconnect topologies in a portable format are promising. Such abstractions naturally complement Chakra's execution graph model, enabling infrastructure-aware performance projection and topology-sensitive scheduling. Paired with Chakra ETs, standard-

ized infrastructure descriptions can further support cross-platform trace reuse, interconnect design comparisons, and feasibility analysis of parallelization strategies under realistic resource constraints. InfraGraph (Keysight) is an emerging effort within the MLCommons Chakra working group along this direction, complementing Chakra with graph-based infrastructure modeling for the trace ecosystem.

6.2.3 Chakra for MLPerf Storage

The MLPerf Storage benchmark (MLPerf) measures how fast storage systems feed the input data to the training model. Its current implementation replaces all compute and communication with a fixed delay and only replays storage operations. This limits fine-grained replay of storage together with compute and communication, and reduces portability across systems. We are working with the MLPerf Storage working group to integrate storage operations into Chakra ETs and enable replay for MLPerf Storage benchmarking.

7 Related Work

The technique of collecting traces to understand system internals and identify bottlenecks is widely adopted and deployed. Google, for instance, collected and released instruction traces from its servers for representative cloud workloads (Ranganathan & Victor). Chakra is a similar effort aimed at ML systems research, specifically for distributed ML systems. In Chakra, we proposed a common schema for ML execution traces. The GOAL format is another post-execution trace that builds around the communication operations, and is used to study both ML and high-performance computing (HPC) workloads (Shen et al., 2025) As ML tasks are often represented as graphs, numerous graph schemas have been proposed. ONNX is one of the most popular graph schemas, designed to facilitate the exchange of models between different ML frameworks. Although ONNX shares similarities with Chakra, such as enabling data exchange between different teams, the objectives of the two differ. ONNX focuses on the exchange of *models* between various frameworks, while Chakra's primary goal is to exchange *execution traces* between different teams. While their purposes differ, ONNX and Chakra can be complementary—models expressed in ONNX can naturally yield execution traces encoded in Chakra. When collected at the pre-execution stage, traces closely resemble the model itself, with additional annotations. These traces are similar to the graphs used in prior autotuning studies to optimize ML model parallelization strategies (Wang et al., 2019; Santhanam et al., 2021; Schaarschmidt et al., 2021). Unity is a representative example: it jointly applies algebraic transformations and parallelization, and introduces a parallel computation graph to support such optimization. In contrast, Chakra is not limited to pre-execution traces; it also supports post-execution traces that capture real system dynamics, such as compute–communication overlap and idle periods, enabling more faithful replay and simulation.

8 Conclusion

In this paper, we present Chakra, an MLCommons-enabled ecosystem for benchmarking and co-designing current and future systems. At its core, Chakra defines an open graph schema, called execution traces, to specify distributed workloads and capture key operations and dependencies. We also develop a complementary set of tools and capabilities for collecting, analyzing, and adopting Chakra execution traces across a wide range of simulators, emulators, and benchmarks. By addressing ecosystem fragmentation and workload-sharing barriers that lead to siloed optimizations and longer time-to-market, Chakra enables a more open and reusable co-design workflow. As future work, we are actively improving trace-size scalability, AI platform infrastructure standardization, and storage extensions.

Acknowledgements

We acknowledge Srinivas Sridharan, Taekyung Heo, Joongun Park, and Brian Coutinho for the initial prototyping of Chakra. We thank Matt Bergeron, Wenyin Fu, Zhaodong Wang, Shengbao Zheng from Meta who helped develop the tooling and vision for early versions of Chakra. We acknowledge Theodor Adrian Badea from Keysight as a co-author for this paper, whose name we unintentionally missed during paper registration. We sincerely thank MLCommons for their role in establishing the Chakra Working Group and the technical inputs. We appreciate the support for Chakra from a large set of community advocates: Shashi Gandham (Meta), Alex Bortok and Ram Periakaruppan (Keysight), Arindam Mallick and Debjyoti Bhattacharjee (IMEC), Chun Liu (ByteDance), Ulf Hanebutte (Marvell), Samantika Sury (HPE), Oana Balmau (McGill University). We appreciate Matthieu Bloch and Aaron Jezghani in helping us leverage the AI Makerspace within the College of Engineering (RRID:SCR_028058) at the Georgia Institute of Technology, provided by the Partnership for an Advanced Computing Environment (PACE) (RRID:SCR_027619), for collecting a library of traces for this effort.

The results and comparisons provided in this paper are meant to showcase the value of the Chakra ecosystem and are not a direct endorsement of any specific vendor's hardware or software products. AMD, the AMD Arrow logo, Instinct, RCCL, and combinations thereof are trademarks of Advanced Micro Devices, Inc. PyTorch, the PyTorch logo and any related marks are trademarks of The Linux Foundation. TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc. NCCL, NeMo and CUDA are trademarks of NVIDIA Corp. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

ASTRA-sim. Astra-sim: Scalable system-level simulation framework for large-scale machine learning systems. https://astra-sim.github.io/, 2025. Accessed: 2025-10-30.

Bambhaniya, A. R., Wu, H., Subramanian, S., Srinivasan, S., Kundu, S., Yazdanbakhsh, A., Elavazhagan, M., Kumar, M., and Krishna, T. Understanding and optimizing multi-stage ai inference pipelines, 2026. URL https://arxiv.org/abs/2504.09775.

Bergeron, M. and Kumar, A. Accelerating AI Hardware NPI - Clusterless Validation of GPUs and Networking. https://youtu.be/-PRs1eVF3nY?si=sYl3P0tSsmEAJOL2, 2025. OCP Global Summit.

Bortok, A. Methodology and Observation of Congestion Control Impact on MoE Training Job Completion Time. https://youtu.be/nLSDrgvu-qw?si=EJnOJ__zB35delA1, 2025. OCP Global Summit.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Dai, D., Deng, C., Zhao, C., Xu, R. X., Gao, H., Chen, D., Li, J., Zeng, W., Yu, X., Wu, Y., Xie, Z., Li, Y. K., Huang, P., Luo, F., Ruan, C., Sui, Z., and Liang, W. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models, 2024. URL https://arxiv.org/abs/2401.06066.

FacebookResearch. Holistic trace analysis. https://github.com/facebookresearch/HolisticTraceAnalysis. Accessed: 2025-09-28.

Feng, L. [PyTorch] Integrate Execution Graph Observer into PyTorch Profiler. URL https://github.com/pytorch/pytorch/pull/75358.

Georgia Institute of Technology. Georgia tech ai makerspace. https://coe.gatech.edu/academics/ai-for-engineering/ai-makerspace, 2026. Accessed: 2026-04-06.

Go, S., Park, J., More, S., Wu, H., Wang, I., Jezghani, A., Krishna, T., and Mahajan, D. Characterizing the efficiency of distributed training: A power, performance, and thermal perspective. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO '25, pp. 626–642, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715730. doi: 10.1145/3725843.3756111. URL https://doi.org/10.1145/3725843.3756111.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Wyatt, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Guzmán, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Thattai, G., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Zhang, J., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Prasad, K., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Lakhotia, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Tsimpoukelli, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Zhang, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Maheswari, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Albiero, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Wang, X., Tan, X. E., Xia, X., Xie, X., Jia, X., Wang, X., Gold-

schlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Srivastava, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Teo, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Dong, A., Franco, A., Goyal, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Liu, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Gao, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Le, E.-T., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Kokkinos, F., Ozgenel, F., Caggioni, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Inan, H., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Zhan, H., Damlaj, I., Molybog, I., Tufanov, I., Leontiadis, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Lam, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Jagadeesh, K., Huang, K., Chawla, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Liu, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Mehta, N., Laptev, N. P., Dong, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Parthasarathy, R., Li, R., Hogan, R., Battey, R., Wang, R., Howes, R., Rinott, R., Mehta, S., Siby, S., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Mahajan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Patil, S., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Deng, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Koehler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wu, X., Wang, X., Wu, X., Gao, X., Kleinman, Y., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Zhao, Y., Hao, Y., Qian, Y., Li, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., Zhao, Z., and Ma, Z. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

Keysight. Infrastructure Graph. URL https://infragraph.dev/.

Keysight Technologies. Keysight ai data center builder. https://github.com/Keysight/kai-dc-builder/releases/download/v1.0.2/Keysight.AI.Data.Center.Builder.Solution.Brief.pdf, 2025. Solution brief. Accessed: 2026-03-31.

Li, M., Shitole, V., Chien, E., Man, C., Wang, Z., Zhang, Y., Krishna, T., Li, P., et al. Layerdag: A layerwise autoregressive diffusion model of directed acyclic graphs for system. In *The 13th International Conference on Learning Representations (ICLR)*, 2024.

Liang, M., Fu, W., Feng, L., Lin, Z., Panakanti, P., Zheng, S., Sridharan, S., and Delimitrou, C. Mystique: Enabling Accurate and Scalable Generation of Production AI Benchmarks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

Liang, M., Kassa, H. T., Fu, W., Coutinho, B., Feng, L., and Delimitrou, C. Lumos: Efficient performance modeling

and estimation for large-scale llm training, 2025. URL https://arxiv.org/abs/2504.09307.

Man, C. symbolic_tensor_graph: A symbolic tensor graph generator for astra-sim. https://github.com/astra-sim/symbolic_tensor_graph, 2024. Accessed: 2025-10-29.

Man, C., Park, J., Wu, H., Xu, H., Sridharan, S., and Krishna, T. Scalable synthesis of llm benchmarks through symbolic tensor graphs. In *Proceedings of the 53rd IEEE/ACM International Symposium on Computer Architecture (ISCA '26)*, 2026.

Meta. Chakra execution traces: Benchmarking network performance optimization. https://engineering.fb.com/2023/09/07/networking-traffic/chakra-execution-traces-benchmarking-network-performance-optimization/, September 2023. Accessed: 2026-03-31.

Minhas, M. S. Computational Graphs in PyTorch and TensorFlow. URL https://towardsdatascience.com/computational-graphs-in-pytorch-and-tensorflow-c25cc40bdcd1. Accessed: 2025-10-01.

MLCommons. Chakra: Advancing benchmarking and co-design for future ai systems. https://mlcommons.org/2023/07/chakra-advancing-benchmarking-and-co-design-for-future-ai-systems/, July 2023. Accessed: 2026-03-31.

MLPerf. MLPerf Storage Benchmark. URL https://mlcommons.org/benchmarks/storage/.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems, 2019. URL https://arxiv.org/abs/1906.00091.

NeMo. NVIDIA NeMo. https://www.nvidia.com/en-us/ai-data-science/products/nemo/. Accessed: 2025-10-01.

Patel, P., Choukse, E., Zhang, C., Shah, A., Íñigo Goiri, Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting, 2024. URL https://arxiv.org/abs/2311.18677.

PyTorch. Kineto: A cpu+gpu profiling library. https://github.com/pytorch/kineto, 2025. GitHub repository.

Ranganathan, P. and Victor, L. Advancing systems research with open-source Google workload traces. URL https://cloud.google.com/blog/topics/systems/workload-traces-for-google-warehouse-scale-computers.

Rashidi, S., Denton, M., Sridharan, S., Srinivasan, S., Suresh, A., Nie, J., and Krishna, T. Enabling compute-communication overlap in distributed deep learning training platforms. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 540–553. IEEE, 2021a.

Rashidi, S., Denton, M., Sridharan, S., Srinivasan, S., Suresh, A., Nie, J., and Krishna, T. Enabling compute-communication overlap in distributed deep learning training platforms. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, pp. 540–553. IEEE Press, 2021b. ISBN 9781450390866. doi: 10.1109/ISCA52012.2021.00049. URL https://doi.org/10.1109/ISCA52012.2021.00049.

Rashidi, S., Won, W., Srinivasan, S., Gupta, P., and Krishna, T. Fred: A wafer-scale fabric for 3d parallel dnn training. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pp. 34–48, 2025.

Research, F. Param. https://github.com/facebookresearch/param. Accessed: 2025-09-28.

Santhanam, K., Krishna, S., Tomioka, R., Harris, T., and Zaharia, M. DistIR: An Intermediate Representation and Simulator for Efficient Neural Network Distribution. *arXiv preprint arXiv:2111.05426*, 2021.

Scala Computing. The scala compute platform. https://www.scalacomputing.com/platform, 2025. Accessed: 2025-10-30.

Schaarschmidt, M., Grewe, D., Vytiniotis, D., Paszke, A., Schmid, G. S., Norman, T., Molloy, J., Godwin, J., Rink, N. A., Nair, V., et al. Automap: Towards Ergonomic Automated Parallelism for ML Models. *arXiv preprint arXiv:2112.02958*, 2021.

Shen, S., Bonato, T., Hu, Z., Jordan, P., Chen, T., and Hoefler, T. ATLAHS: An Application-centric Network Simulator Toolchain for AI, HPC, and Distributed Storage, 2025. URL https://arxiv.org/abs/2505.08936.

Sridharan, S., Heo, T., Feng, L., Wang, Z., Bergeron, M., Fu, W., Zheng, S., Coutinho, B., Rashidi, S., Man, C., and Krishna, T. Chakra: Advancing performance benchmarking and co-design using standardized execution traces, 2023. URL https://arxiv.org/abs/2305.14516.

vLLM. vllm github. https://github.com/vllm-project/vllm. GitHub repository, accessed March 30, 2026.

vLLM. [BugFix] Use late binding to avoid zmq port conflict race conditions. https://github.com/vllm-project/vllm/pull/30520, December 2026. GitHub pull request #30520, accessed 2026-03-31.

Wang, F., Chen, G., Zhang, W., and Rompf, T. Parallel Training via Computation Graph Transformation. In *Proceedings of the International Conference on Big Data (Big Data)*, pp. 3430–3439. IEEE, 2019.

Wang, X., Li, Q., Xu, Y., Lu, G., Li, D., Chen, L., Zhou, H., Zheng, L., Zhang, S., Zhu, Y., Liu, Y., Zhang, P., Qian, K., He, K., Gao, J., Zhai, E., Cai, D., and Fu, B. Simai: unifying architecture design and performance tuning for large-scale large language model training with scalability and precision. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-46-5.

Wareing, R. and Graf, T. The Need for Speed: The Story of How We Achieved Industry Leading TTP. https://youtu.be/-PRs1eVF3nY?si=sYl3P0tSsmEAJOL2, 2025. @Scale: Systems & Reliability 2025.

Won, W., Elavazhagan, M., Srinivasan, S., Gupta, S., and Krishna, T. Tacos: Topology-aware collective algorithm synthesizer for distributed machine learning. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 856–870, 2024a. doi: 10.1109/MICRO61859.2024.00068.

Won, W., Rashidi, S., Srinivasan, S., and Krishna, T. LIBRA: Enabling Workload-Aware Multi-Dimensional Network Topology Optimization for Distributed Training of Large AI Models. In *Proceedings of the 2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '24)*, pp. 205–216, 2024b. doi: 10.1109/ISPASS61541.2024.00028.

Yoo, J., Cowan, M., Du, Z., Man, C., Sridharan, S., and Krishna, T. Flint: Compiler enabled cluster-free design space exploration for distributed ml, 2026a. URL https://arxiv.org/abs/2604.17550.

Yoo, J., Lao, C., Cao, L., Lantz, B., Yu, M., Krishna, T., and Sharma, P. Towards easy and realistic network infrastructure testing for large-scale machine learning. *arXiv preprint arXiv:2504.20854*, 2026b. URL https://arxiv.org/abs/2504.20854. Accessed: 2026-03-06.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024. URL https://arxiv.org/abs/2401.09670.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code and instructions for Chakra trace generation, linking, conversion, and analysis. In particular, it includes: (i) utilities to Chakra collected host-side and device-side traces collected from nemo, as shown in Sec. 3 (ii) the Chakra Trace Linker and Converter, which merge host and device traces into a unified dependency graph and emit a standardized Chakra Execution Trace (ET) Sec. 3.1.1, (iii) downstream use cases of Chakra traces for execution-trace analysis, including Astra-sim simulation and Sec. 5.4 (iv) plotting scripts to reproduce Figures 6, 7, 8, and 12.

The artifact evaluates two representative workflows. First, it demonstrates trace linking and conversion using pre-collected traces from NVIDIA NeMo. Second, it demonstrates a downstream use case by using Chakra traces to drive Astra-sim simulations different communication topologies.

A.2 Artifact check-list (meta-information)

- **Program:** Chakra Trace Linker/Converter, trace processing utilities, Astra-sim integration scripts, and plotting scripts
- **Compilation:** Python package installation for Chakra; Astra-sim built in Docker container
- **Model:** Mixtral 8×7B trace used for simulation
- **Data set:** Pre-collected NVIDIA NeMo traces (PyTorch Observer + Kineto)
- **Run-time environment:** Linux; Docker; Python 3.10+
- **Hardware:** CPU-only machine
- **Metrics:** ET validity, dependency statistics, simulated runtime under varied communication topologies, and reproduced analysis figures
- **Output:** Chakra ETs, Astra-sim results, and reproduced figures
- **How much disk space required (approximately)?:** 10 GB
- **How much time is needed to prepare workflow (approximately)?:** around 2 hours to link chakra-enabled nemo traces and convert it to chakra et that can be consumed by downstream simulator. After conversion, it took around 10-20 minutes to run the Astra-sim simulation.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache-2.0

A.3 Description

A.3.1 How to access

The artifact is publicly available both as a GitHub repository and as an archived Zenodo release. It includes the Chakra trace processing pipeline, example traces, Astra-sim configurations, and the scripts needed to reproduce the results reported in the paper. Detailed setup and usage instructions are provided in the README under the mlsys26 directory of the Chakra repository (https://github.com/mlcommons/chakra/tree/mlsys26) and in the corresponding Zenodo archive (https://doi.org/10.5281/zenodo.19636323).

A.3.2 Hardware dependencies

All evaluations can be executed on a CPU-only machine. The trace linking and conversion pipeline operates on pre-collected trace files, and the downstream Astra-sim experiments run entirely in simulation.

A.3.3 Inputs

The artifact includes pre-collected NVIDIA NeMo traces collected via Chakra-enabled PyTorch profiling (Execution Graph Observer for host and Kineto for device). These traces serve as the inputs to the Chakra trace linking and conversion pipeline. The artifact also includes processed traces and configuration files used for simulation and the details are in the README.

A.3.4 Evaluation workflow

The artifact evaluation consists of three steps:

- **Trace linking and conversion.** The provided NeMo host and device traces are linked to construct a unified dependency graph, which is then converted into a standardized Chakra ET.
- **Downstream simulation.** The generated Chakra ET is used as input to Astra-sim to simulate execution of Mixtral 8×7B. By varying the communication topology, users can observe the corresponding differences in simulated runtime.
- **Figure reproduction.** The plotting scripts operate on the processed traces and simulation outputs to reproduce Figures 6, 7, 8, and 12 from the paper.

A.3.5 Expected results

Successful execution of the artifact should produce:

- A valid Chakra ET successfully generated from the provided NeMo traces that can be used for various downstream analysis
- Astra-sim simulation logs under different communication topologies, used for Figure 12
- Reproduced figures comparable to Figures 6, 7, 8,