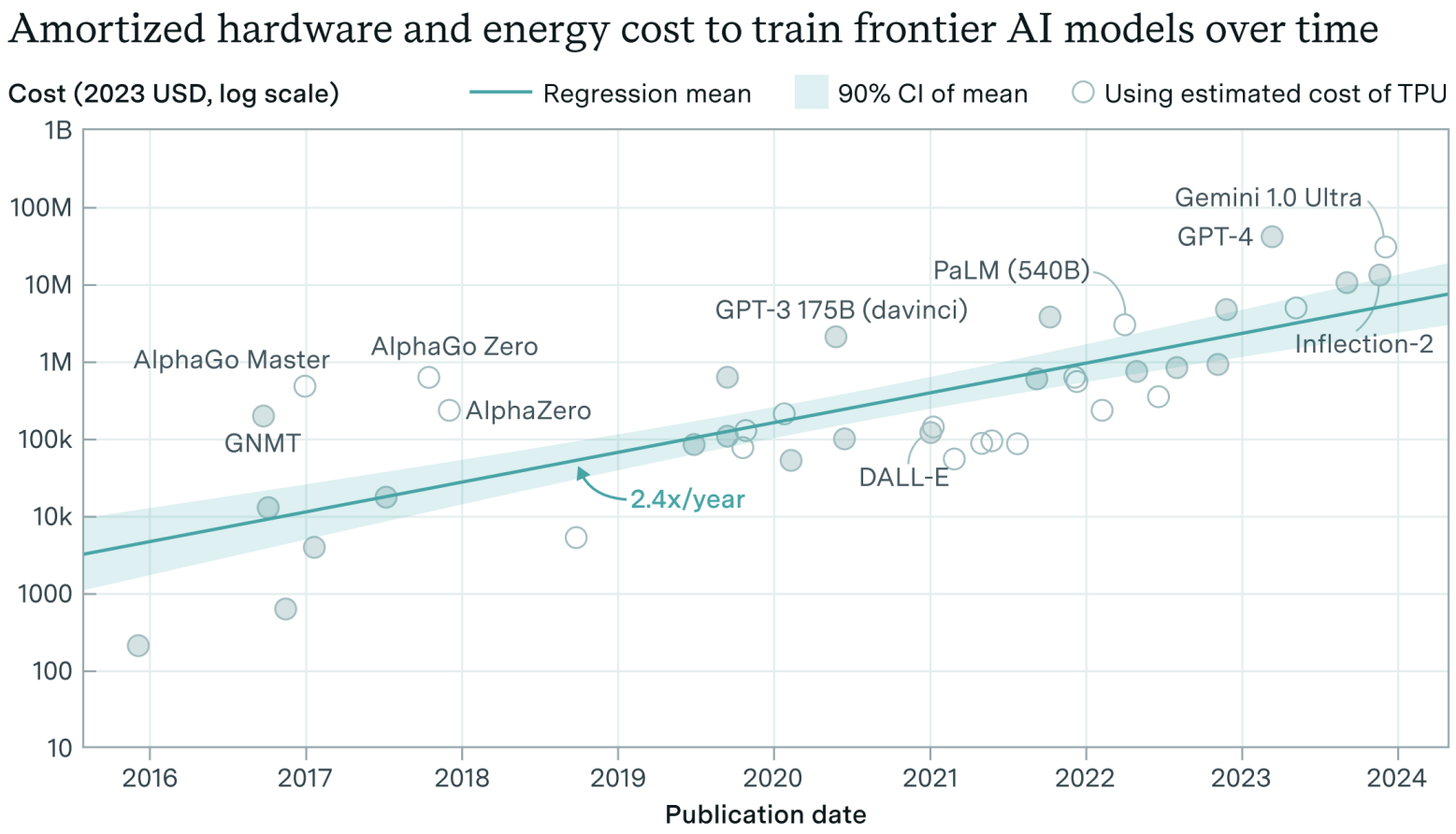
CheckWork: Enabling Trace-Driven Analysis of Checkpointing Overhead in Distributed ML Training

Eldar Hasanov[1], Adel Sefiane[1 2], Alireza Farshin[2], Marios Kogias[1]

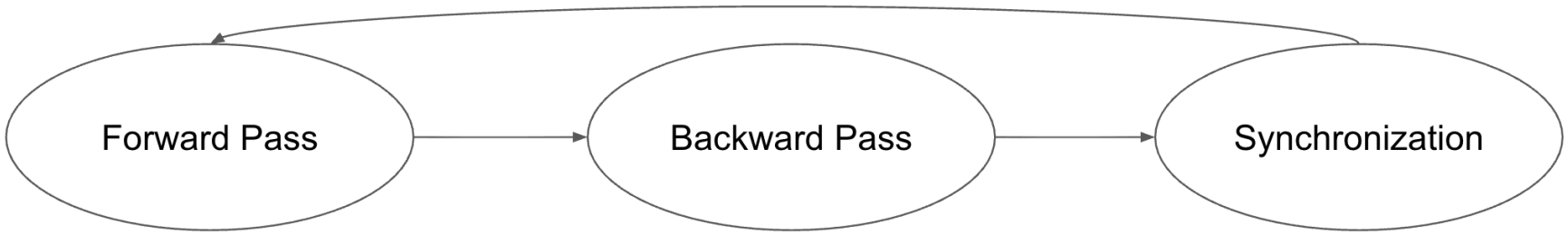
IMPERIAL[1] NVIDIA[2]

AI training is becoming more expensive



Source: Cottier et al., *The Rising Costs of Training Frontier AI Models*, arXiv:2405.21015 (2025)

LLM Training 101



Compute layer outputs

Produce predictions

Compute loss

Compute gradients

Aggregate gradients

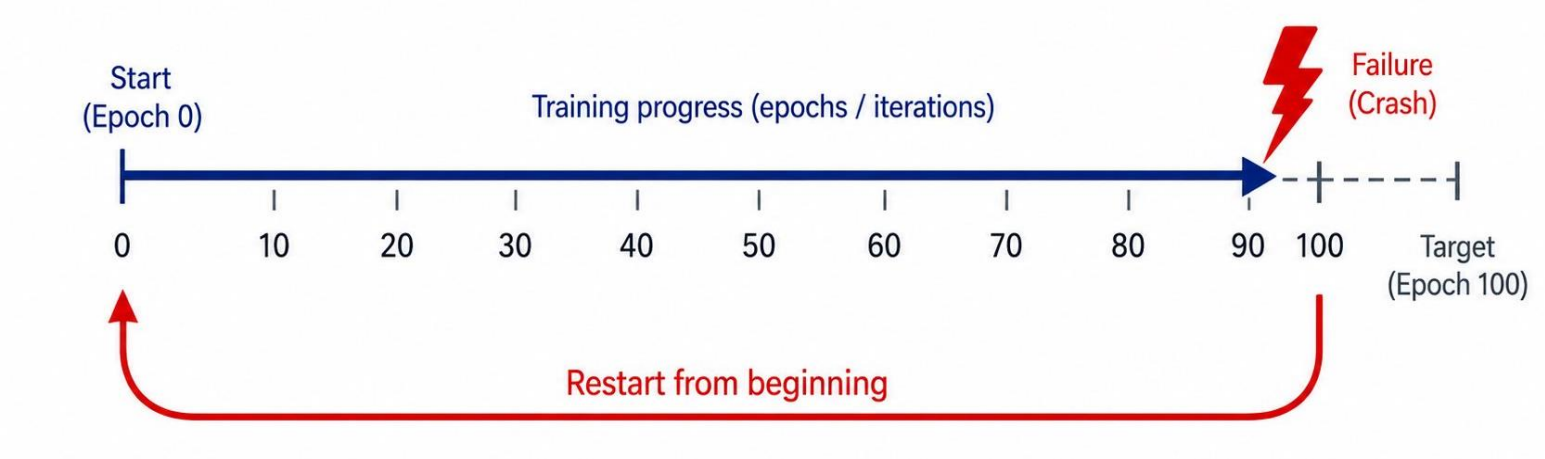
Update parameters

Communication

What can go wrong?

Large-model training is long and expensive.

A *failure* can erase hours or days of computation



At Scale, Failure Is the Norm

Example:
Llama 3 pre-training on 16,384 GPUs showed **419** failures in **54** days (every **3.1** hours).

According to *Bhardwaj et al.*:

1 Failure = **4,096** hours of recomputation

~$15M in wasted compute

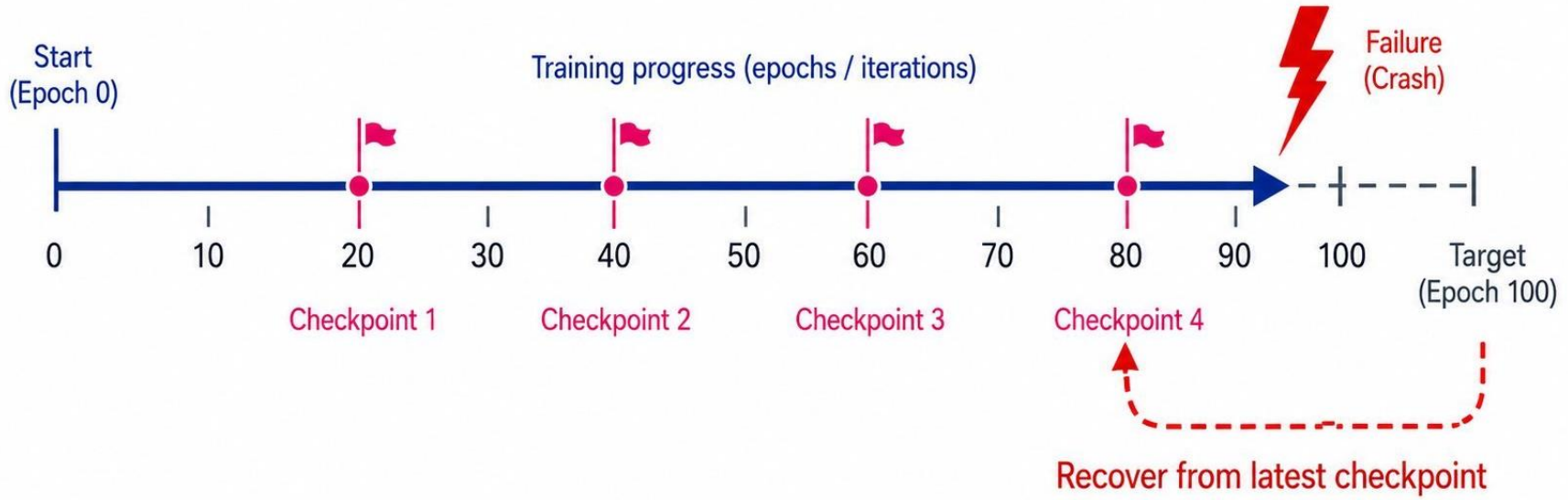
Common failure types:

- GPU or accelerator faults
- Memory errors
- Network or communication failures
- Software crashes and bugs
- Storage or I/O failures

Sources:
Llama Team, *The Llama 3 Herd of Models* (2024);
Bhardwaj et al., *Checkmate*, USENIX NSDI (2026).

Checkpointing Saves The Day

Checkpointing periodically saves the model state to memory or remote storage, allowing training to resume from the latest saved state after a failure.



Reliability

Full snapshots

High frequency

Remote location

Multiple copies

On critical path

Performance

Partial snapshots

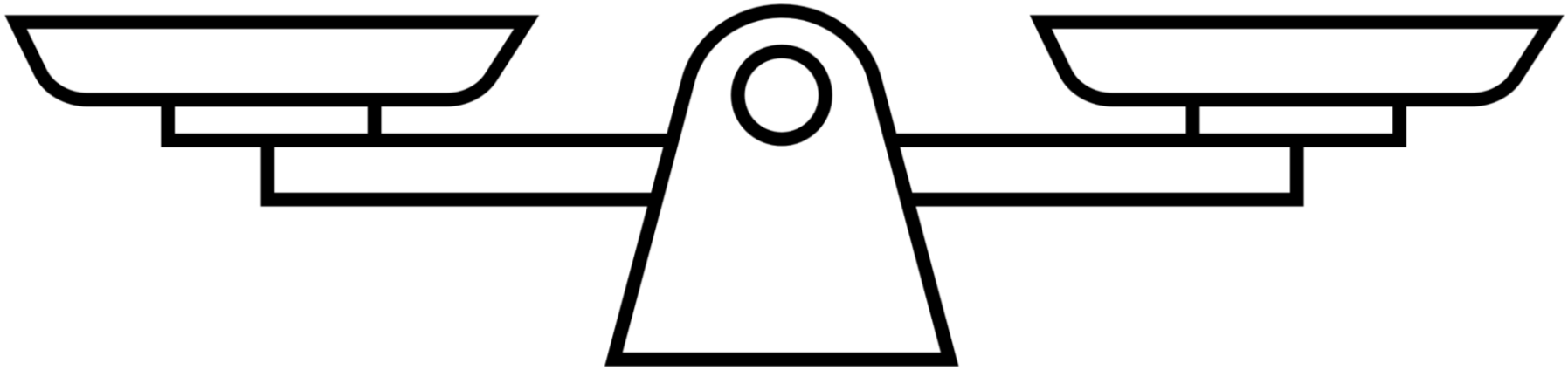
Low frequency

Local / in-memory

Single copy

Off critical path

Reliability **Performance**



Checkpointing strategy directly affects training!

How have checkpointing strategies been evaluated?

Systems from literature	**Compute environment**
CheckFreq, PCcheck	Single-node GPU servers
Gemini, IncrCP, Universal Checkpointing	Cloud and research GPU clusters
Check-N-Run, ByteCheckpoint	Production-scale GPU clusters
DeepFreeze, DataStates-LLM	Leadership HPC systems

Issue: Large-scale training infrastructure is extremely expensive

Simulations as an alternative

Physical testbeds

- Expensive and difficult to access
- Experiments are hard to reproduce
- Only a limited number of configurations can be tested

Simulation

- Inexpensive and repeatable
- Enables large design-space exploration
- What-if analysis across topology, bandwidth, model size, and parallelism

Notable simulators:

- **ASTRA-sim 2.0** *(Won et al., ISPASS '23)*
- **Multiverse** *(Gui et al., NSDI '25)*
- **SimAI** *(Wang et al., NSDI '25)*

But what workload should the simulator execute?

Simulations as an alternative

Physical testbeds

- Expensive and difficult to access
- Experiments are hard to reproduce
- Only a limited number of configurations can be tested

Simulation

- Inexpensive and repeatable
- Enables large design-space exploration
- What-if analysis across topology, bandwidth, model size, and parallelism

Notable simulators:

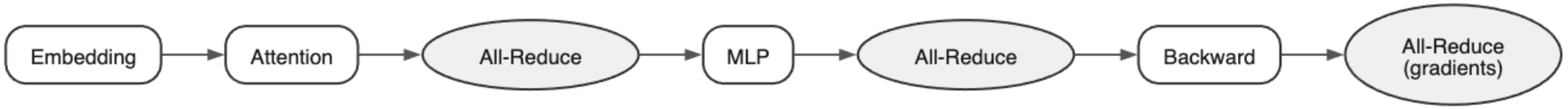
- **ASTRA-sim 2.0** *(Won et al., ISPASS ’23)*
- **Multiverse** *(Gui et al., NSDI ’25)*
- **SimAI** *(Wang et al., NSDI ’25)*

Chakra Execution Traces (ETs)

Open graph schema for standardizing workload specification

Allows representing training workloads as a **DAG** of operations

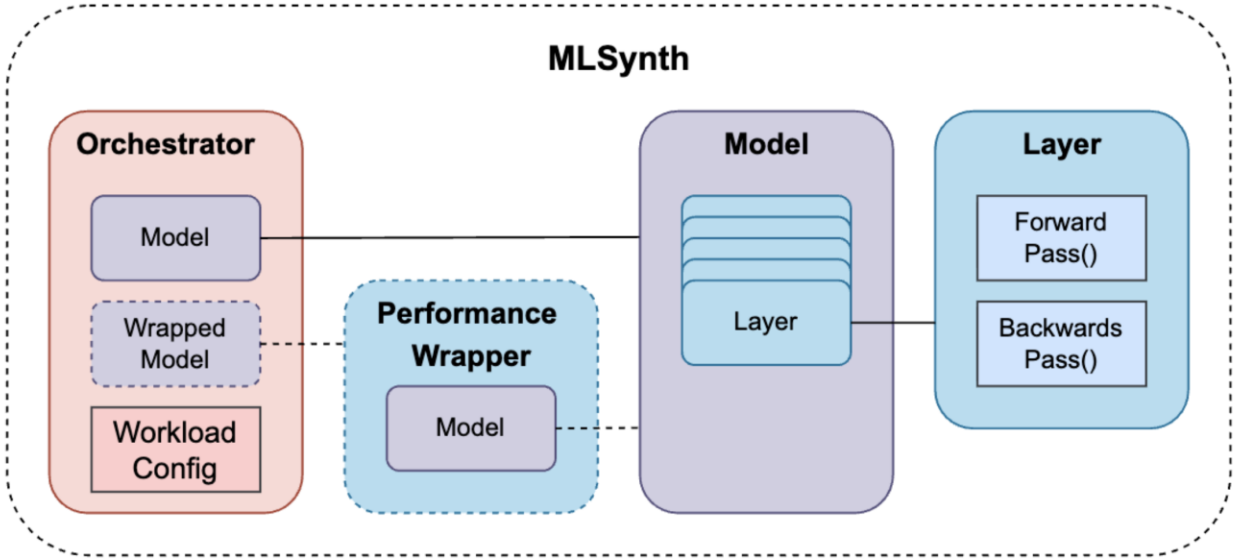
Allows *replaying* the workload in simulations



How to obtain Chakra ETs?

Solution:
Generate synthetic traces

Example: MLSynth
https://github.com/NetMLSim/MLSynth



Inexpensive to produce
Can test fictional scenarios

Lack checkpointing semantics

Why Is Checkpointing Overhead Hard to Evaluate?

Issue:
Existing execution traces model training operations only, **not** checkpointing.

Result:
Evaluating checkpointing strategies still requires costly GPU testbeds

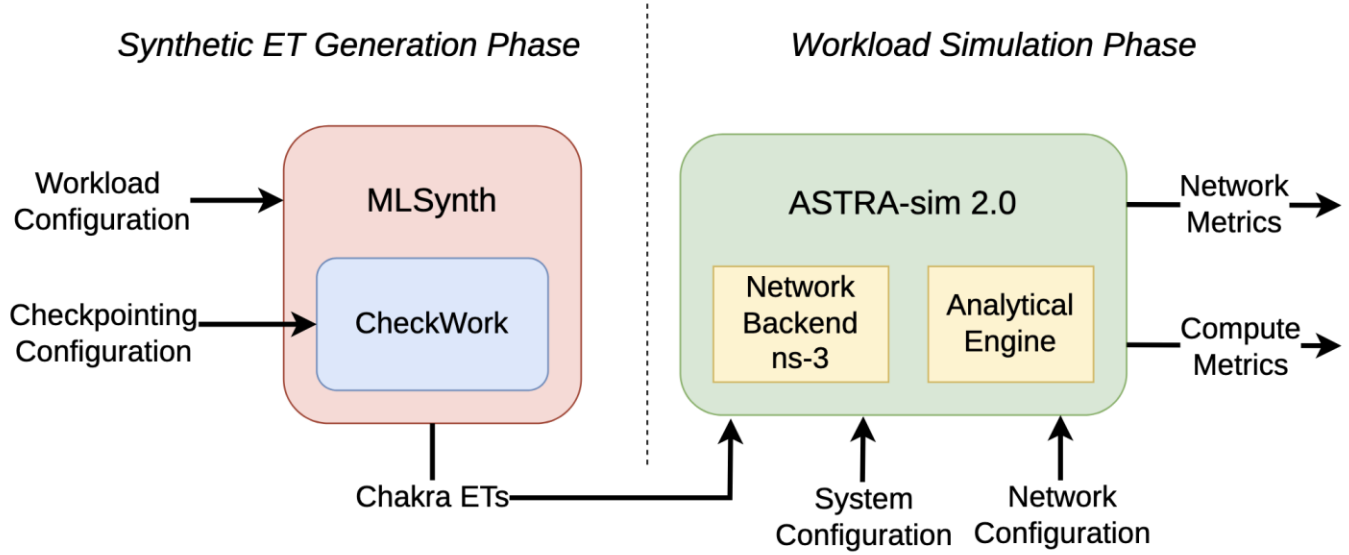
Research gap:
There are no tools to accessibly evaluate checkpointing strategies

?

Solution

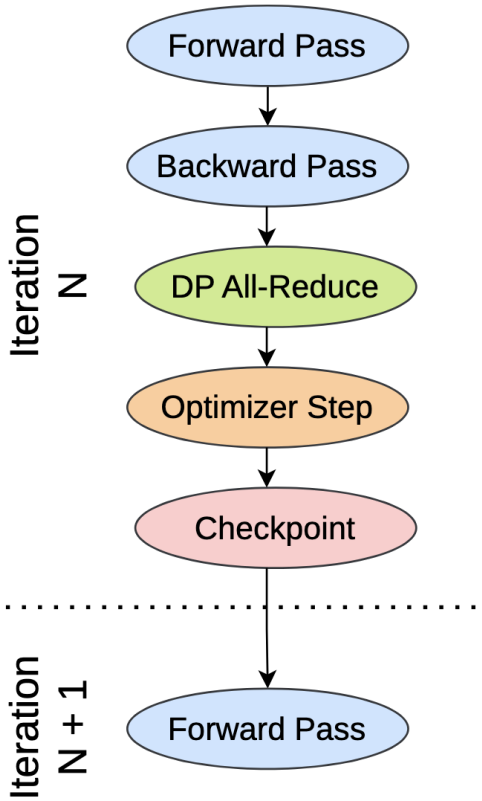
CheckWork: Checkpoint Aware Traces

Framework to integrate checkpointing semantics in Chakra ETs

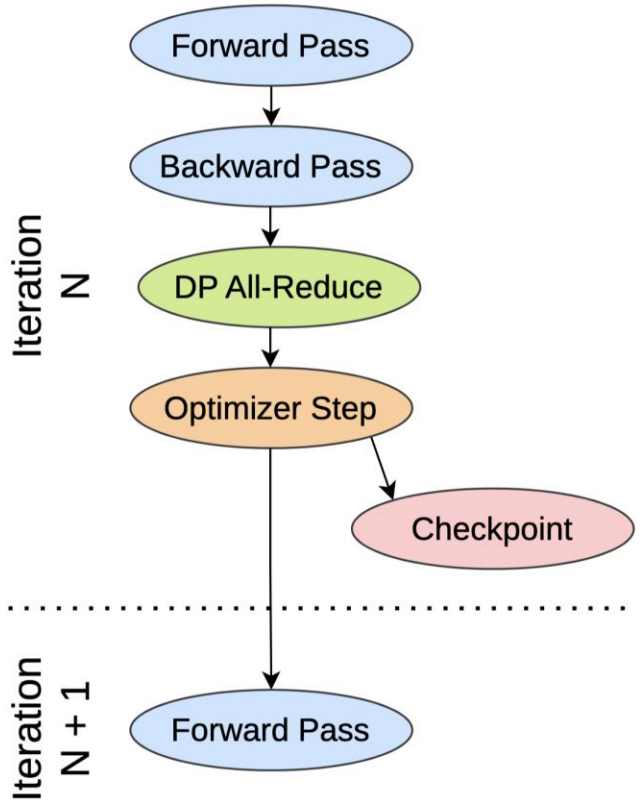


1. Takes checkpointing configurations and ML workload descriptions as input
2. Generates checkpoint-aware Chakra ETs by injecting checkpoint operations
3. Runs directly on ASTRA-sim 2.0 and other Chakra-compatible simulators

CheckWork Generates Chakra ETs



Synchronous Checkpoint



Asynchronous Checkpoint

Configurable Checkpoint Model

Analytical model of checkpoint volume:

$$V = \frac{P}{t\,p}\, b\, \alpha\, s$$

V : checkpoint volume per GPU/rank (bytes)
P : total model parameters
t : tensor-parallel degree
p : pipeline-parallel degree
b : bytes per parameter
α : extra training-state multiplier
s : synthetic model scaling factor

Table 1: Configuration parameters controlling checkpoint behaviour in CheckWork.

Parameter	Description
`mode`	Checkpoint strategy: `sync`, `async`, `remote_sync`, or `remote_async`.
`checkpoint_every_n`	Checkpoint frequency in iterations.
`state_multiplier`	Multiplier used to approximate full training state size (model + optimiser).
`overhead_multiplier`	Scales per-stage checkpoint compute cost and duration.
`kickoff_micros`	Duration of the checkpoint kickoff node (μs) in async and remote modes.
`max_inflight`	Cap on concurrent background writes; `1` serialises, `0` disables the cap.
`remote_target`	In remote modes, destination: `ring`, `rank0`, or `storage`.
`snapshot_multiplier` / `snapshot_micros`	Snapshot cost scale (bytes × multiplier) and optional duration override (μs).
`persist_multiplier` / `persist_micros`	Persist cost scale (bytes × multiplier) and optional duration override (μs).

Evaluation

How do we evaluate CheckWork?

Computational Overhead

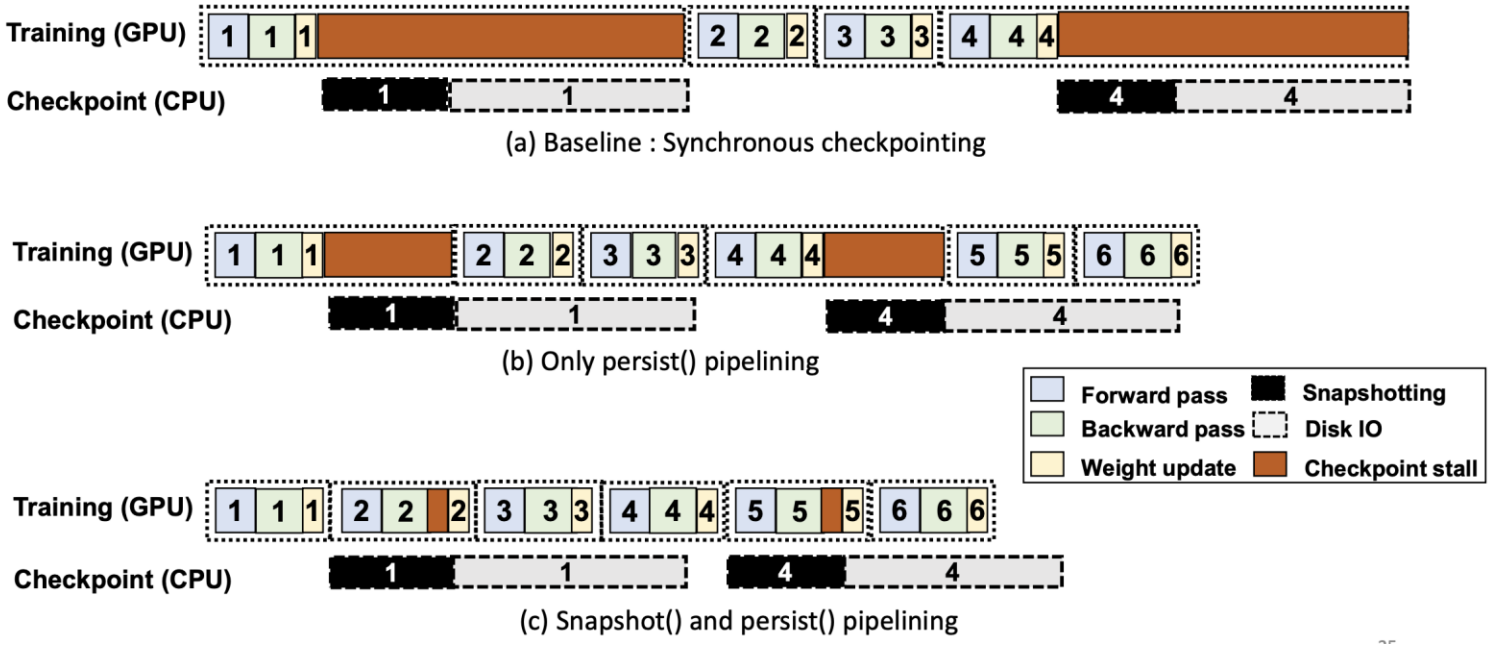
Can CheckWork accurately demonstrate compute-level overhead caused by checkpointing?

Network-level Effects

Can CheckWork accurately reproduce network-level behavior of previous systems and validate new ones?

CheckFreq (FAST'21)

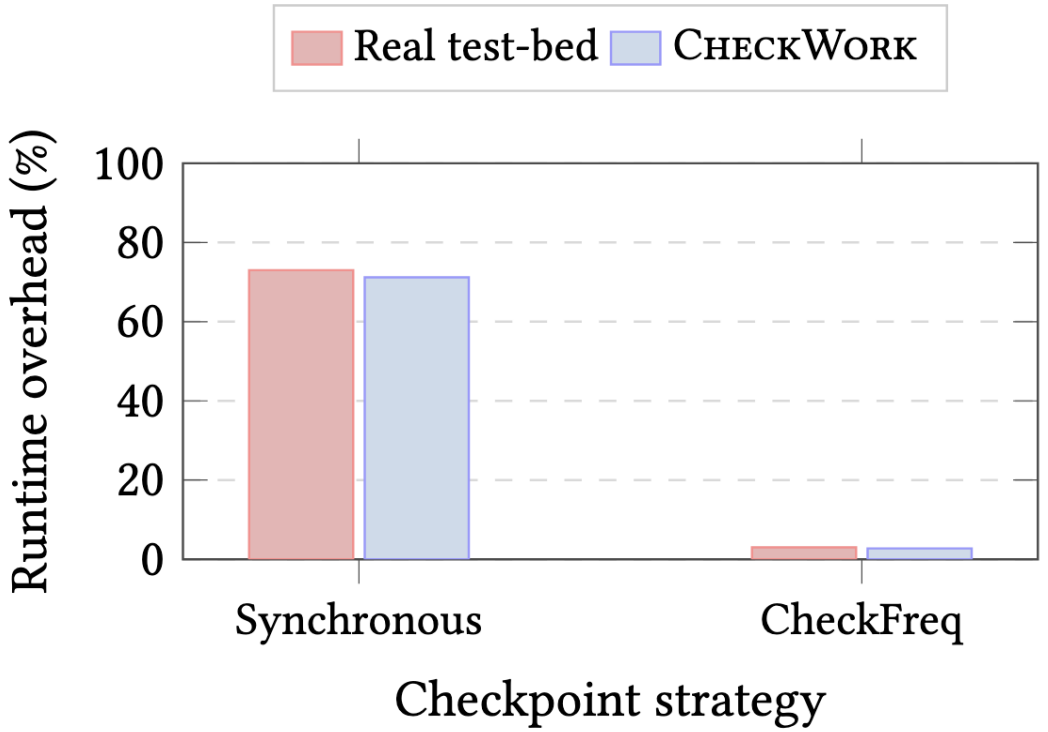
Splits checkpointing into a fast **GPU→CPU snapshot** + background **CPU→storage write**. Persistence is overlapped with training; only the snapshot remains on the critical path.



Source: *Mohan et al., "CheckFreq: Frequent, Fine-Grained DNN Checkpointing," USENIX FAST '21.*

CheckFreq (FAST'21)

CheckWork successfully reproduced the results!



Our setup:

- ASTRA-sim 2.0 analytical engine
- <5 minutes on an Apple M5 laptop
- Peak memory: 1.5 GiB

CheckFreq original setup:

- 8-GPU server with 24 CPU cores
- V100 and GTX 1080 Ti hardware

GEMINI (SOSP ‘23)

Problem:

Remote storage is slow. Checkpoint transfers can interfere with training communication.

Idea:

Store checkpoint shards in the **memory of other training nodes** instead of immediately writing them to remote storage.

Mechanism:

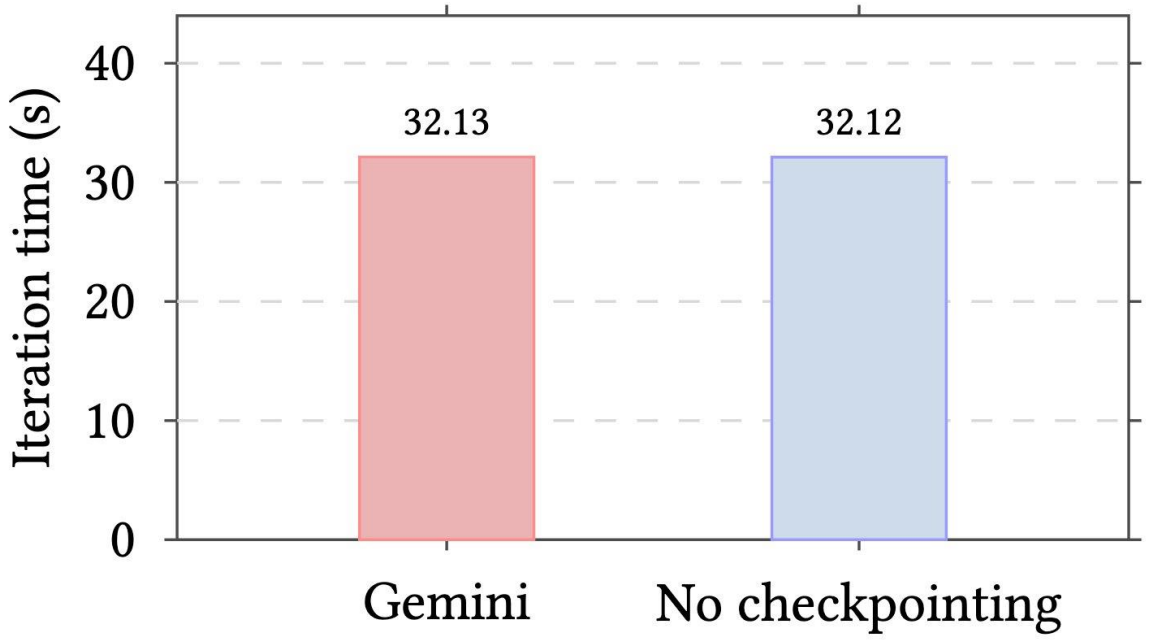
Transfer checkpoints asynchronously.

Give **training traffic higher network priority** than checkpoint traffic.

Source: *Wang et al., “GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints,” ACM SOSP ’23.*

GEMINI (SOSP ‘23)

CheckWork successfully reproduced the results!



Our setup:

- 32 GPUs
- Oversubscribed three-tier fat-tree
- 3D: TP = 4, PP = 2, DP = 4
- ASTRA-sim 2.0 with ns-3 backend
- Prioritization via traffic classes

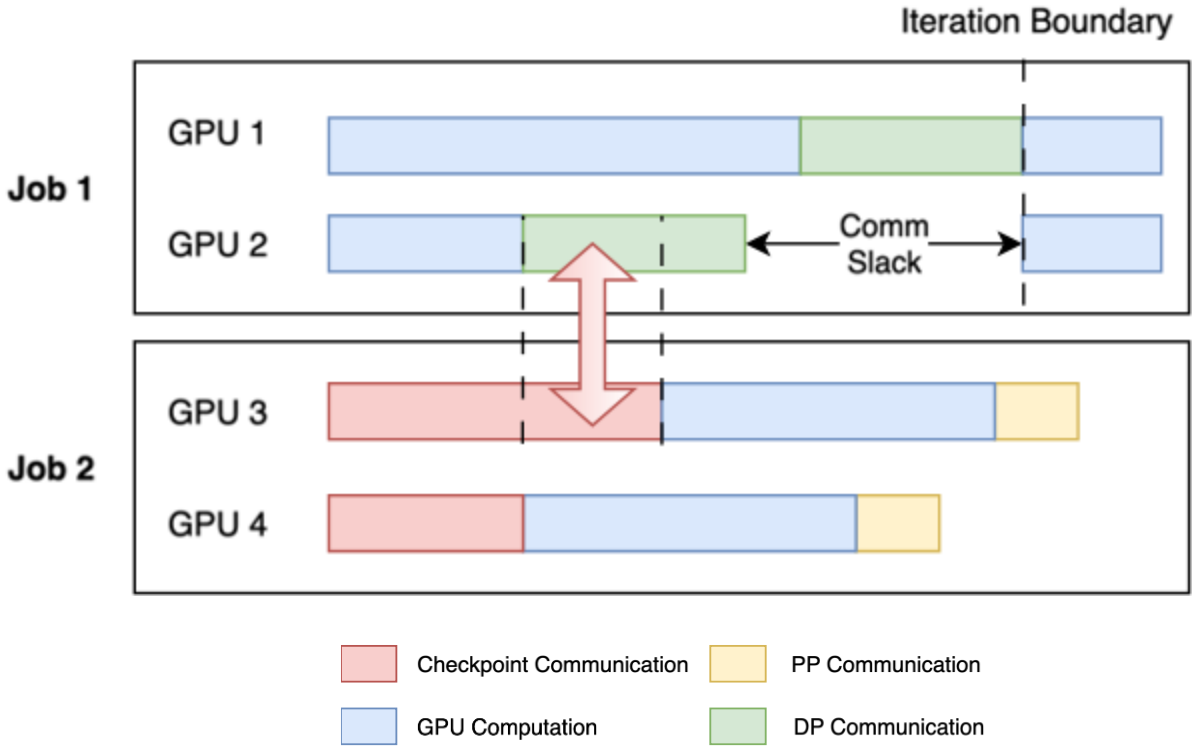
GEMINI original setup:

- 16 AWS p4d.24xlarge instances
- 8× A100 40 GB per instance
- 128 A100 GPUs total

What-if analysis

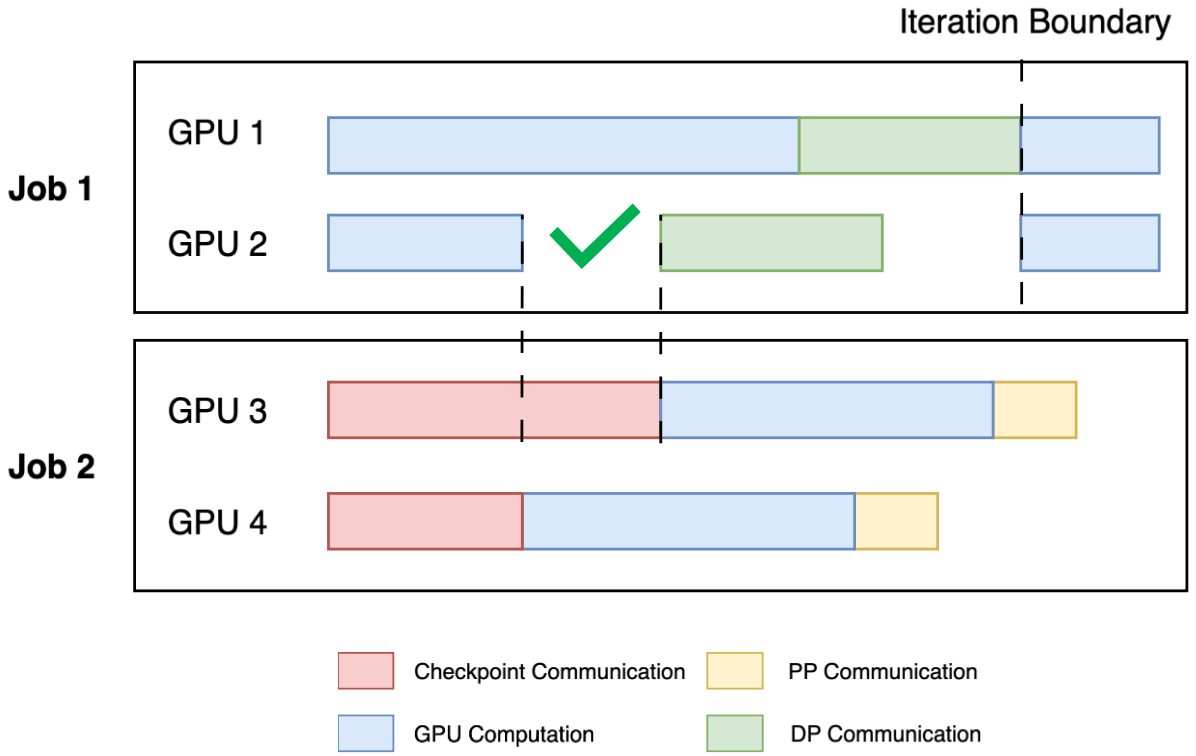
Analyzing multi-job traffic

Problem: In multi-job environments, checkpoint traffic and training traffic can overlap to create interference

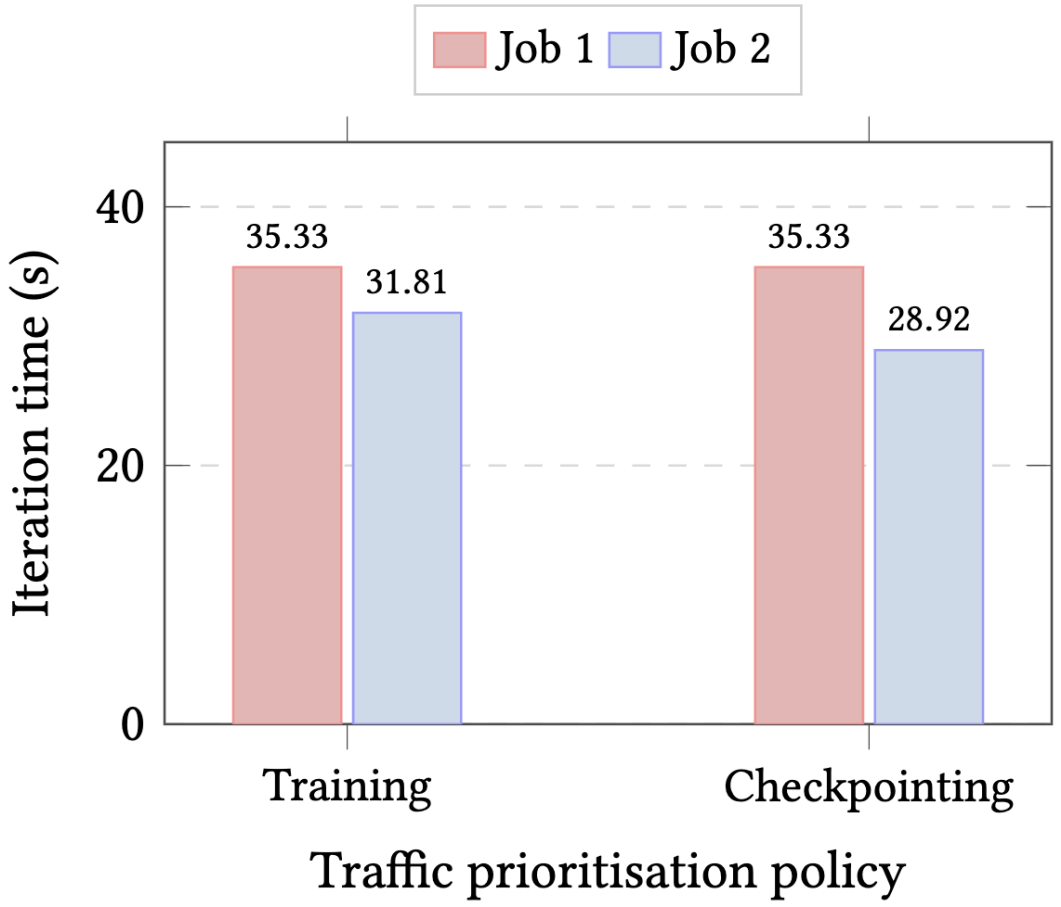


Analyzing multi-job traffic

Hypothesis: In presence of *non-critical communication slack*, checkpoint traffic should be prioritized over training traffic



Results



Setup:

- 2 Transformer jobs
- 16 GPUs each, in the same pod
- Oversubscribed 3-tier fat-tree
- ASTRAsim2.0 with ns-3 backend

Implication:

In-multi job scenarios checkpoint traffic prioritization can sometimes improve job completion latency

Conclusion

As part of this work we:

1. Introduced *CheckWork* – a framework to synthesize Chakra ETs with checkpoint semantics
2. Evaluated trace fidelity to results from CheckFreq and Gemini
3. Used CheckWork to create and simulate multi-job training scenarios to present a case for checkpoint traffic prioritization

https://github.com/NetMLSim/checkwork