

MCCL: Fault-Tolerant Collective Communication

Ben Carver
Network.AI @ Meta

∞ Meta

Scaling Meta's Frontier Models

LLAMA2

70B Parameters
2T Tokens

8x

LLAMA3

405B Parameters
15.6T Tokens

8x

LLAMA4

288B/2T Parameters
30T Tokens



Scaling Meta's Frontier Models

LLAMA2

70B Parameters
2T Tokens

8x

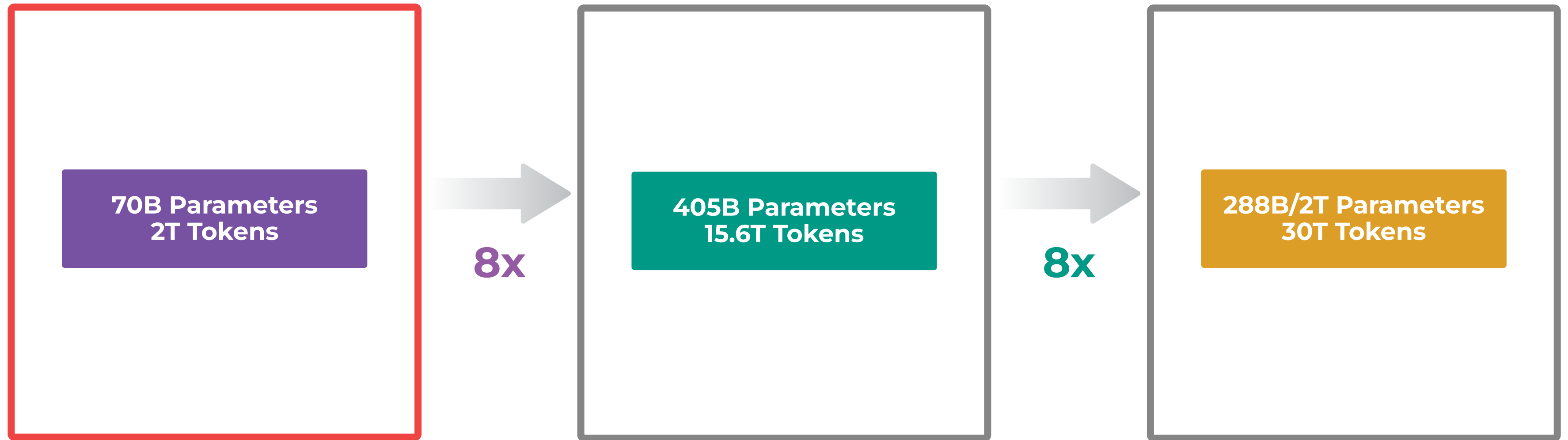
LLAMA3

405B Parameters
15.6T Tokens

8x

LLAMA4

288B/2T Parameters
30T Tokens



Scaling Meta's Frontier Models

LLAMA2

70B Parameters
2T Tokens

8x

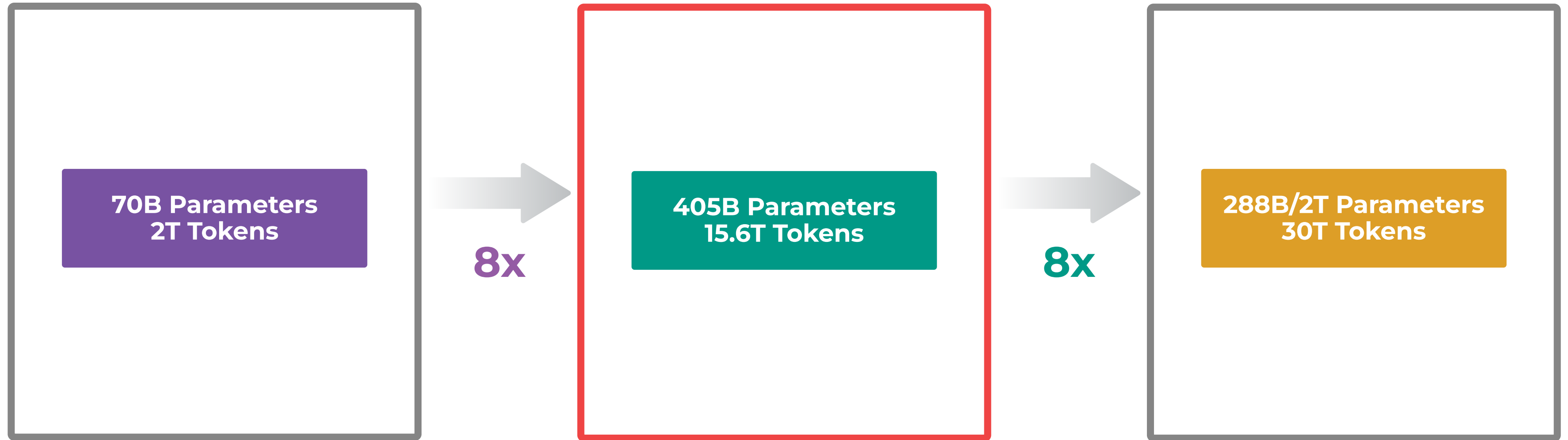
LLAMA3

405B Parameters
15.6T Tokens

8x

LLAMA4

288B/2T Parameters
30T Tokens



Scaling Meta's Frontier Models

LLAMA2

70B Parameters
2T Tokens

8x

LLAMA3

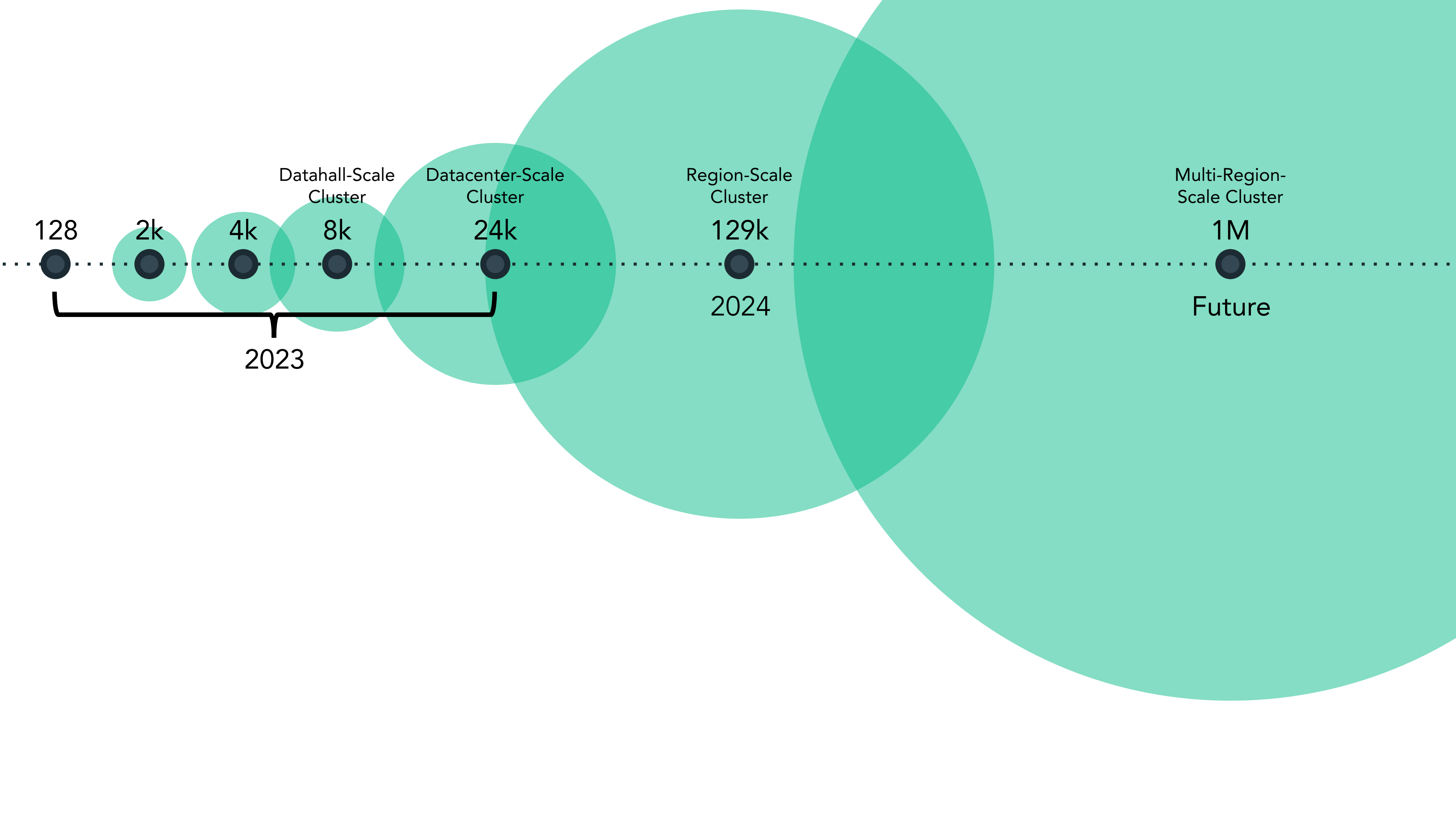
405B Parameters
15.6T Tokens

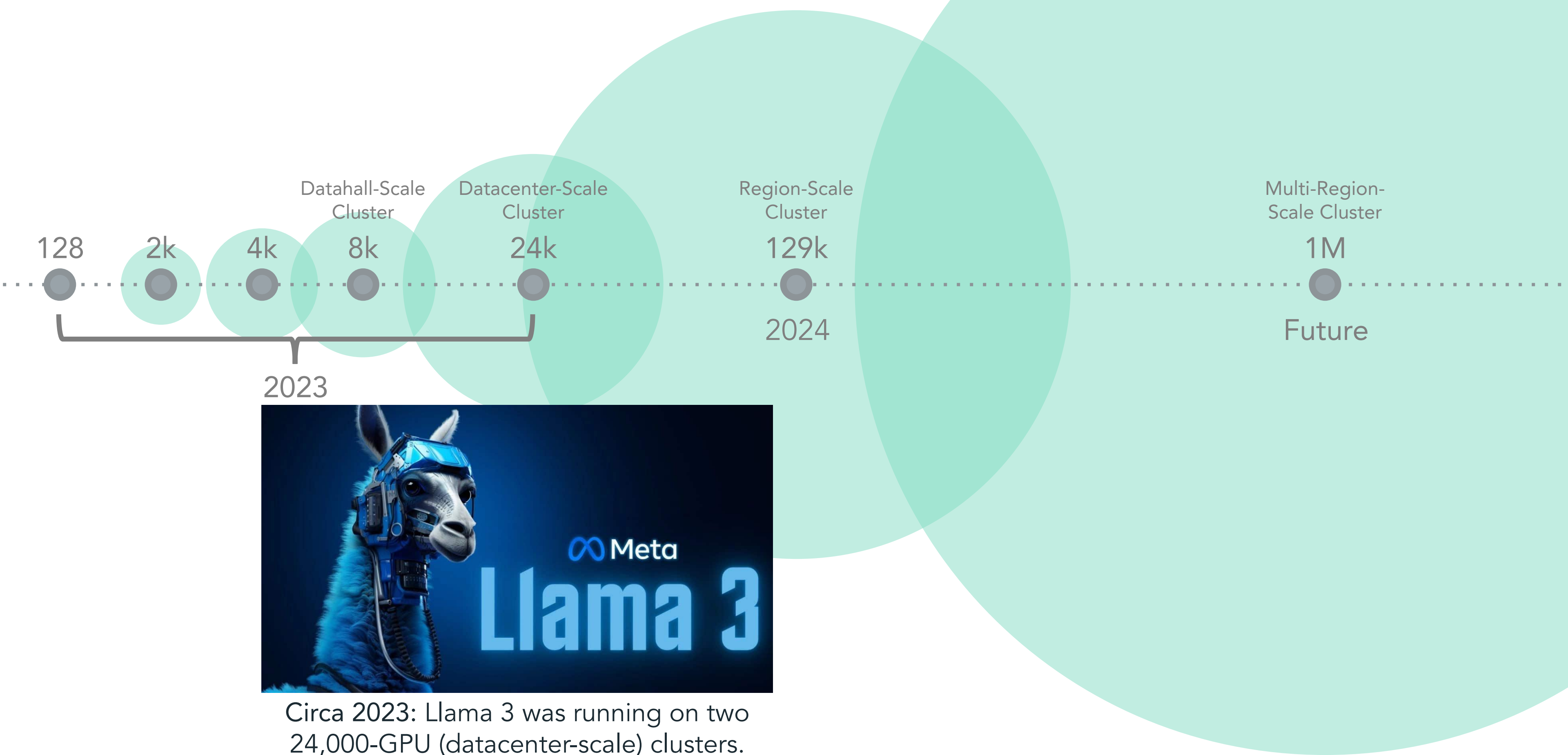
8x

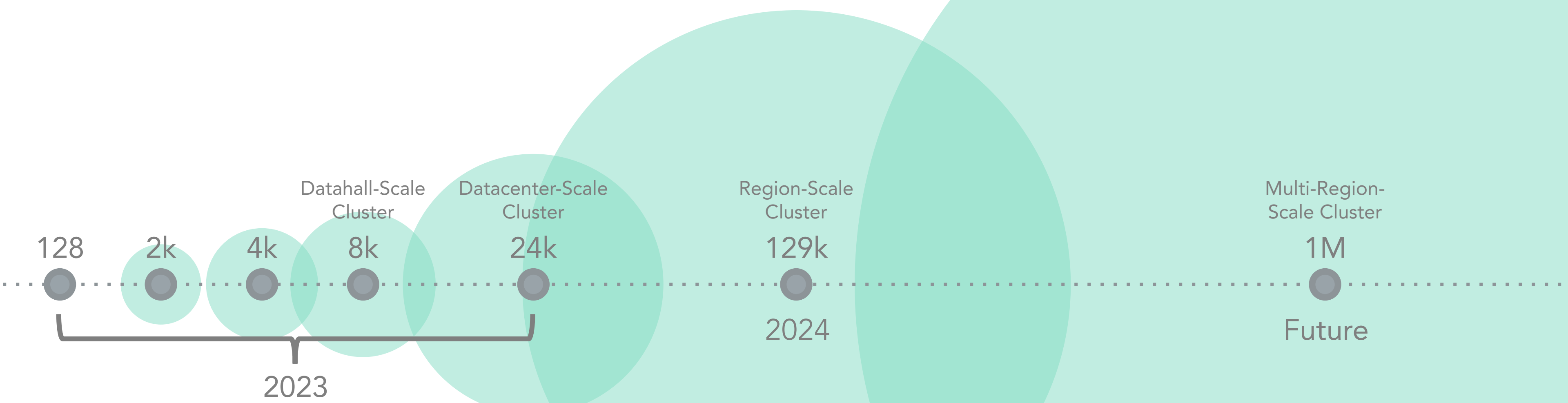
LLAMA4

288B/2T Parameters
30T Tokens





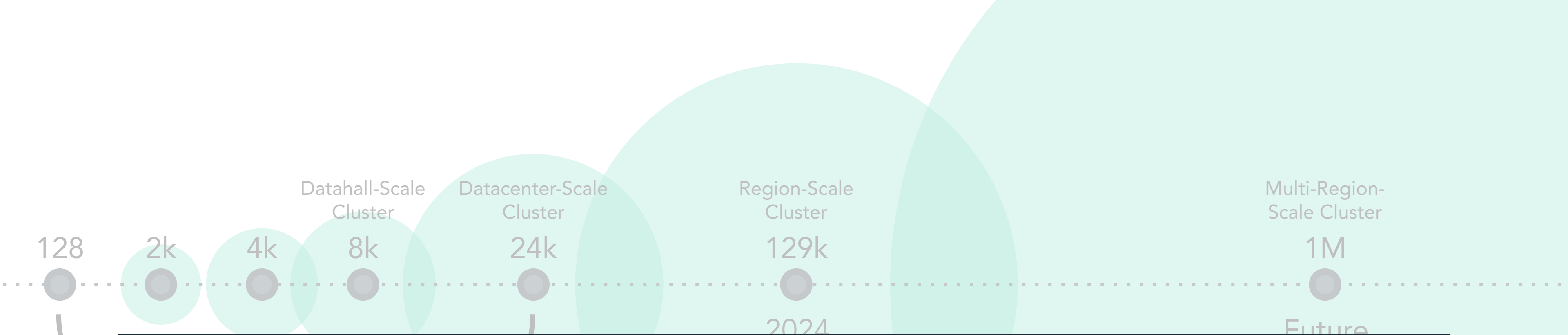




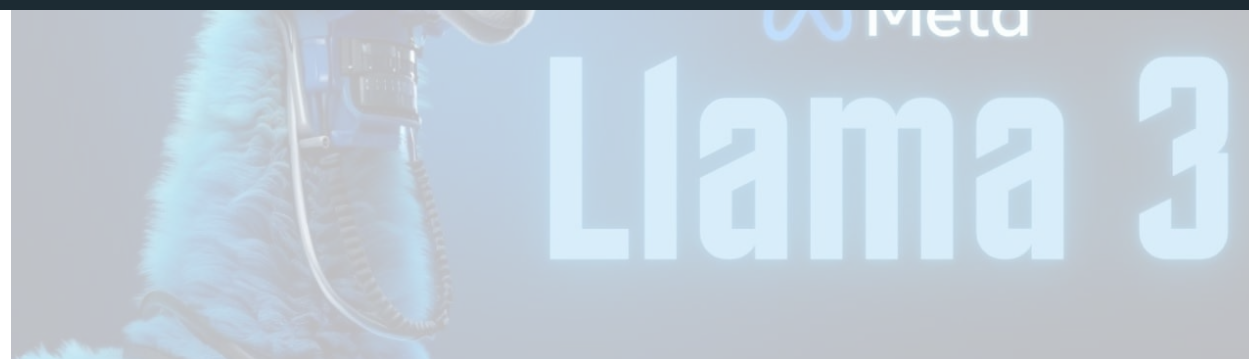
Circa 2023: Llama 3 was running on two 24,000-GPU (datacenter-scale) clusters.



Circa 2024: Llama 4 was running on over 100,000 GPUs – a regional-scale cluster.



**For future models, we're projecting
1,000,000+ GPUs**

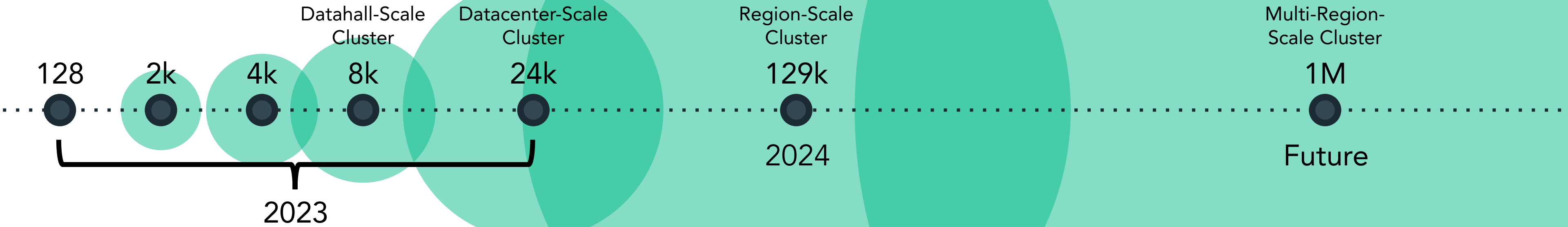


Circa 2023: Llama 3 was running on two 24,000-GPU (datacenter-scale) clusters.



Circa 2024: Llama 4 was running on over 100,000 GPUs – a regional-scale cluster.

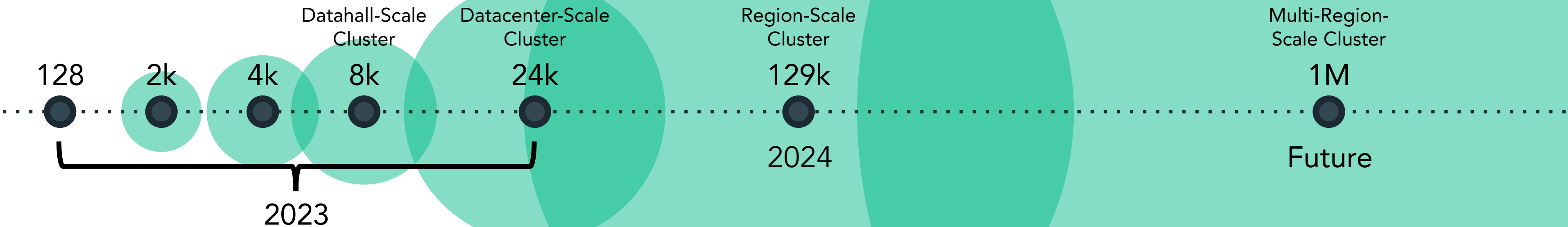
At scale, failure is routine



Tens of thousands of GPUs per job

Failures occur every few minutes

At scale, failure is routine



Tens of thousands of GPUs per job

Failures occur **every few minutes**

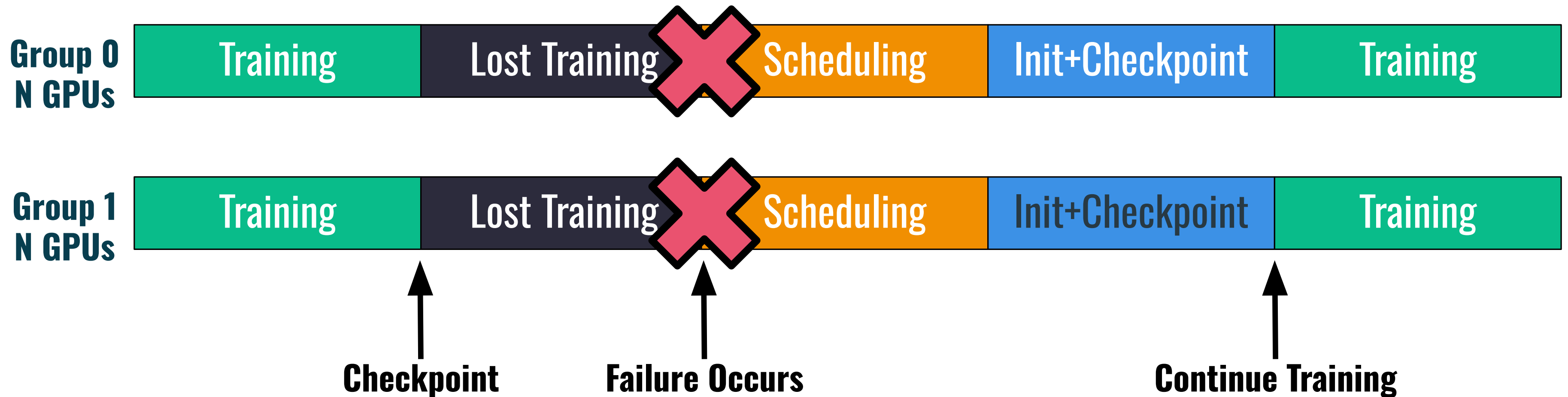
GPUs overheat. Links drop. Nodes die.

Runs last **days or weeks**, so failures compound



What static membership costs you

- 1 GPU fails → every rank stops and whole job restarts from checkpoint
- Tens of minutes lost, per failure → thousands of GPU-hours per day at 100k scale



Introducing MCCL

- **M**eta **C**ollective **C**ommunication **L**ibrary.
- A **high-performance** software library engineered specifically to power **large-scale, fault-tolerant** GPU training across **next-generation infrastructure**.
- Enables rapid iteration for Meta

Introducing MCCL

- Meta Collective Communication Library.
- A high-performance library engineered specifically to power **large-scale, fault-tolerant** GPU training across next-generation infrastructure.
- Enables rapid iteration for Meta

Meta's **latest frontier models**, including **MUSE WONDER**, are trained using **MCCL**!

MCCL Overview

01 / RESILIENCE

Fault-Tolerant Collectives

Built atop dynamic communicators that can seamlessly **shrink and grow** without interrupting cluster stability.

MCCL Overview

01 / RESILIENCE

Fault-Tolerant Collectives

Built atop dynamic communicators that can seamlessly **shrink and grow** without interrupting cluster stability.

02 / HARDWARE COVERAGE

GPU & CPU Support

Comprehensive coverage across heterogenous hardware:

- **GPU:** Built upon Primitives, our novel and efficient **IBGDA-based** collective library.
- **CPU:** Full CPU collectives integration for **easy Control Plane operations.**

MCCL Overview

01 / RESILIENCE

Fault-Tolerant Collectives

Built atop dynamic communicators that can seamlessly **shrink and grow** without interrupting cluster stability.

02 / HARDWARE COVERAGE

GPU & CPU Support

Comprehensive coverage across heterogenous hardware:

- **GPU:** Built upon Primitives, our novel and efficient **IBGDA-based** collective library.
- **CPU:** Full CPU collectives integration for **easy Control Plane operations**.

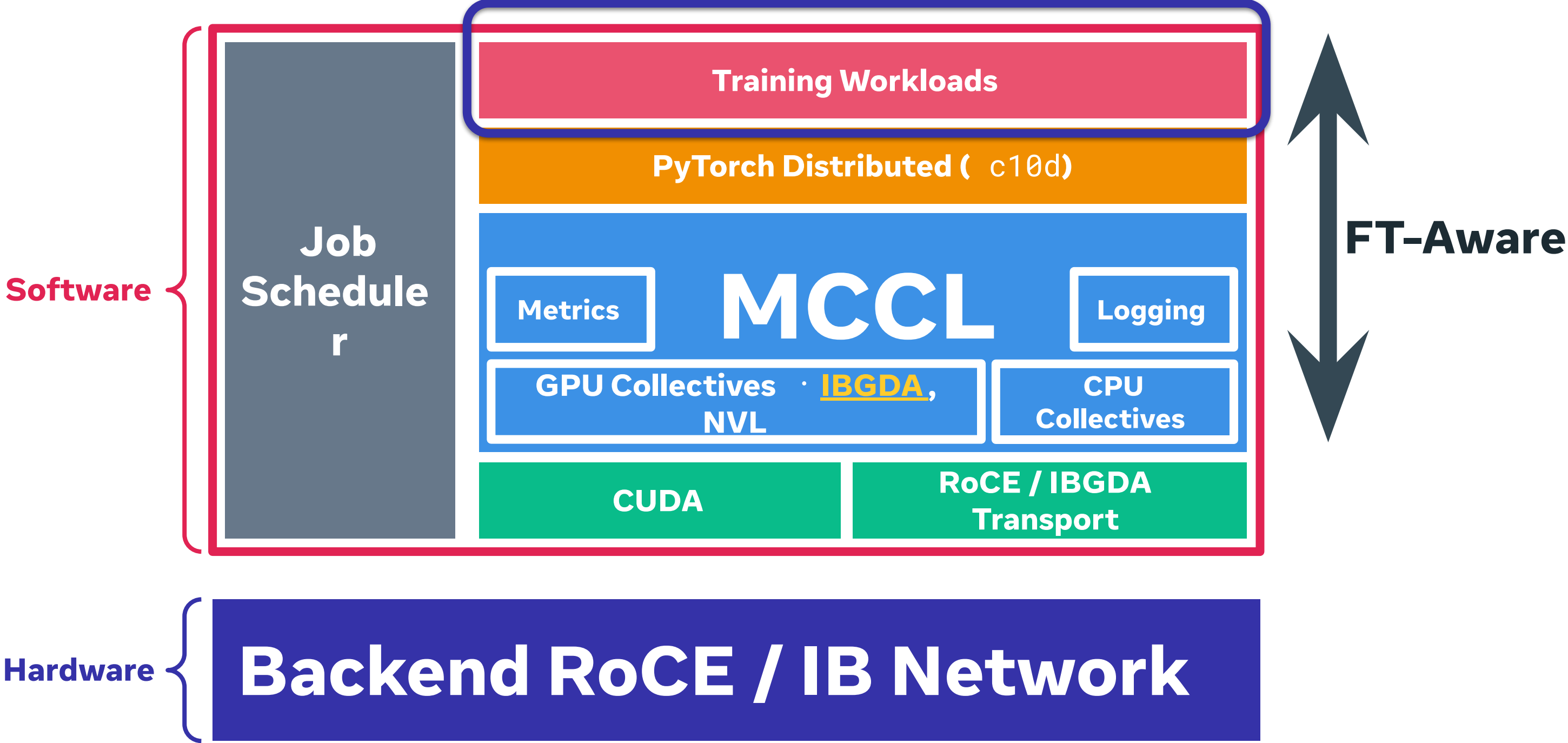
03 / INTEGRATION LAYER

Flexible APIs

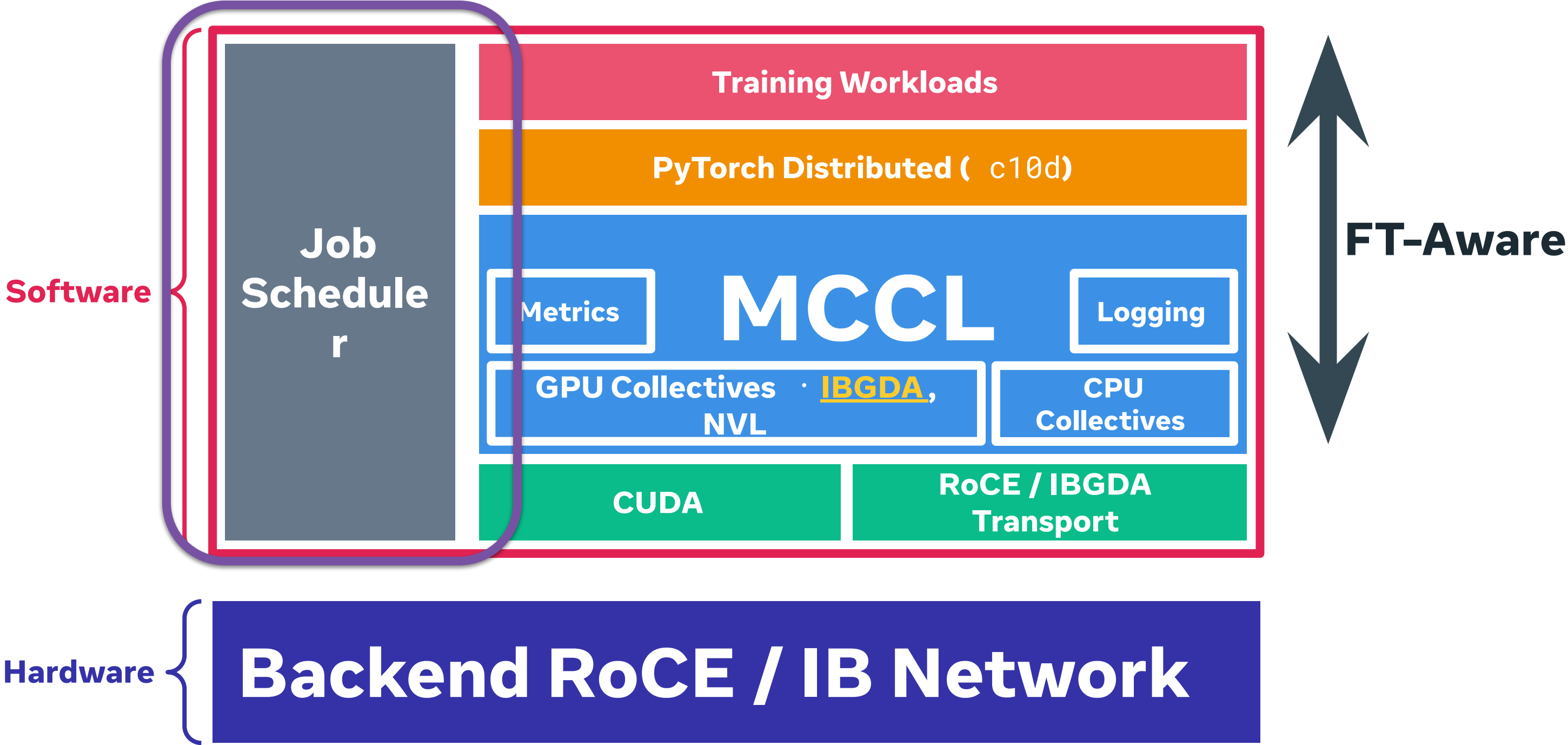
Exposed across standard PyTorch entry points:

- Primary surface: **PyTorch Distributed c10d**.
- Seamless **NCCL adapter** layer.

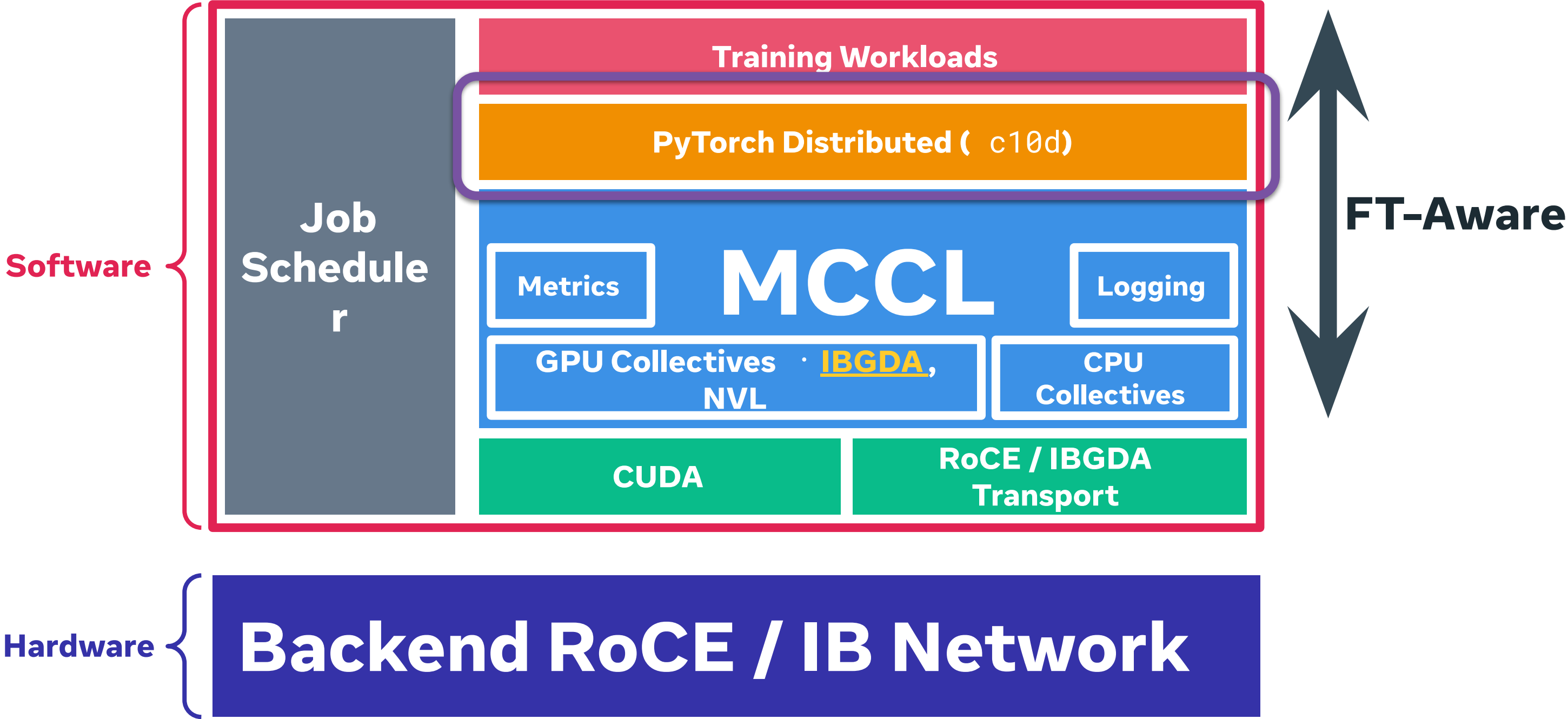
Software Stack



Software Stack

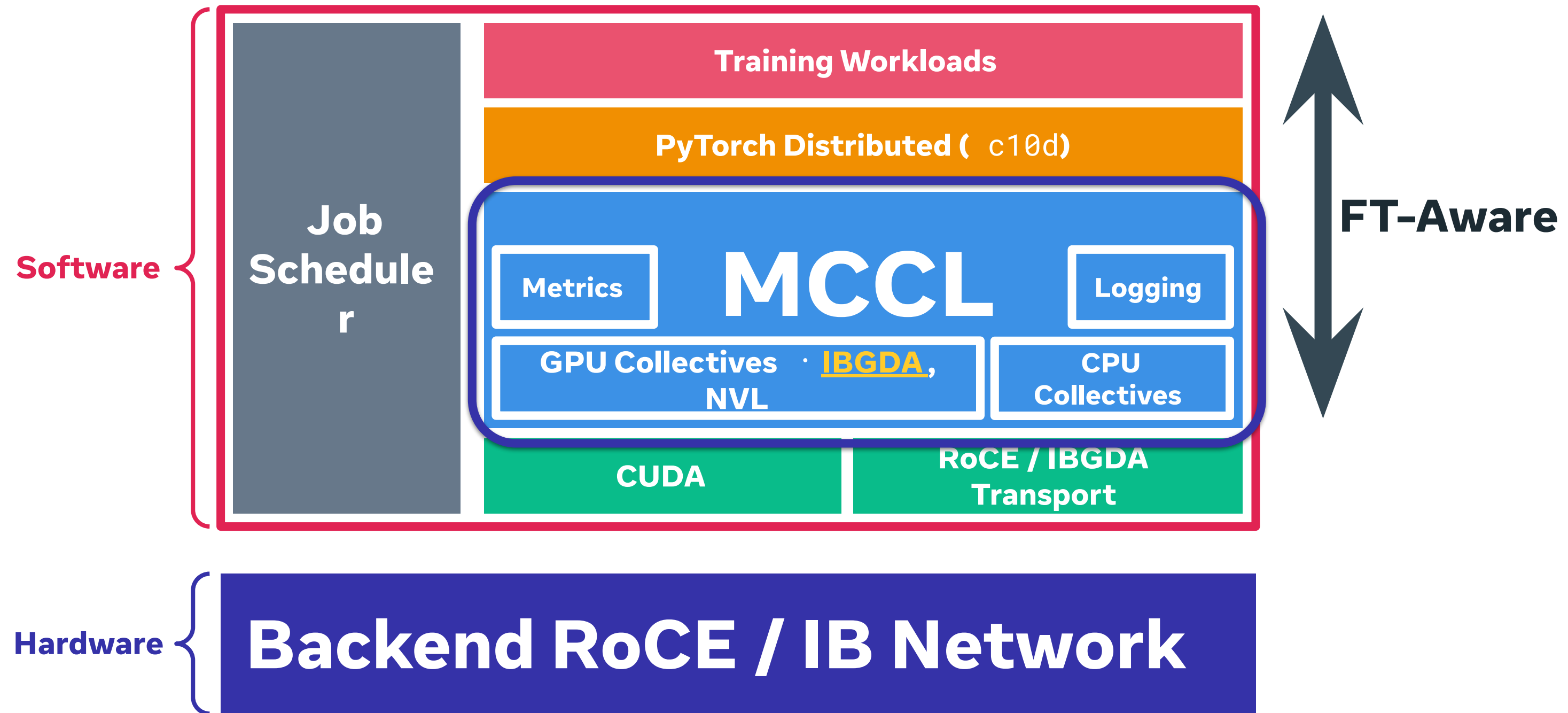


Software Stack



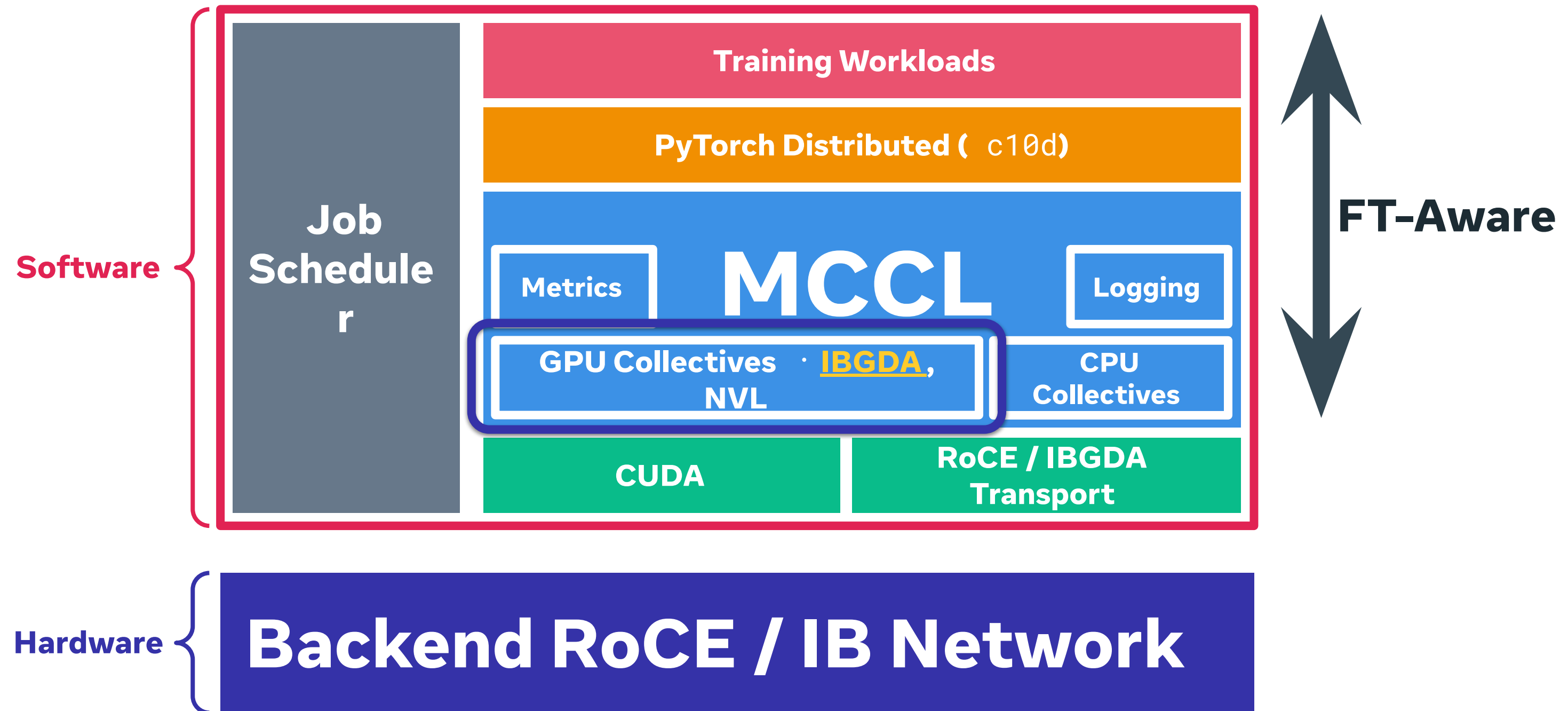
Internal modules: CPU collectives, control plane operations, logging, metrics, etc.

Software Stack



PRIMITIVES: MCCL's native **IBGDA-first** GPU collectives library (also supports NVL).

Software Stack

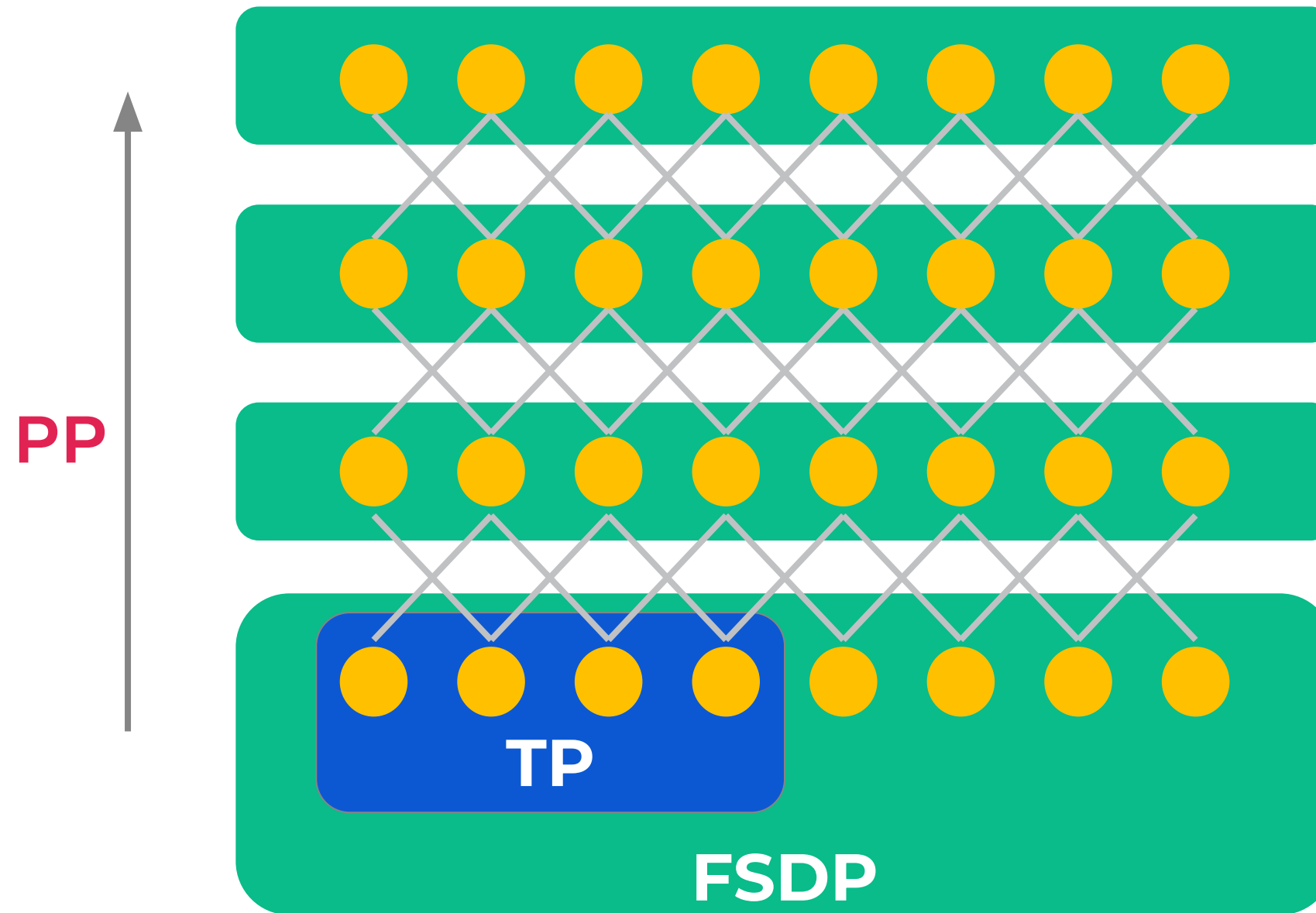


Old parallelism setup

Old parallelism setup

**Model sharded across
ALL GPUs.**

**Losing a single shard →
no longer have full
state**



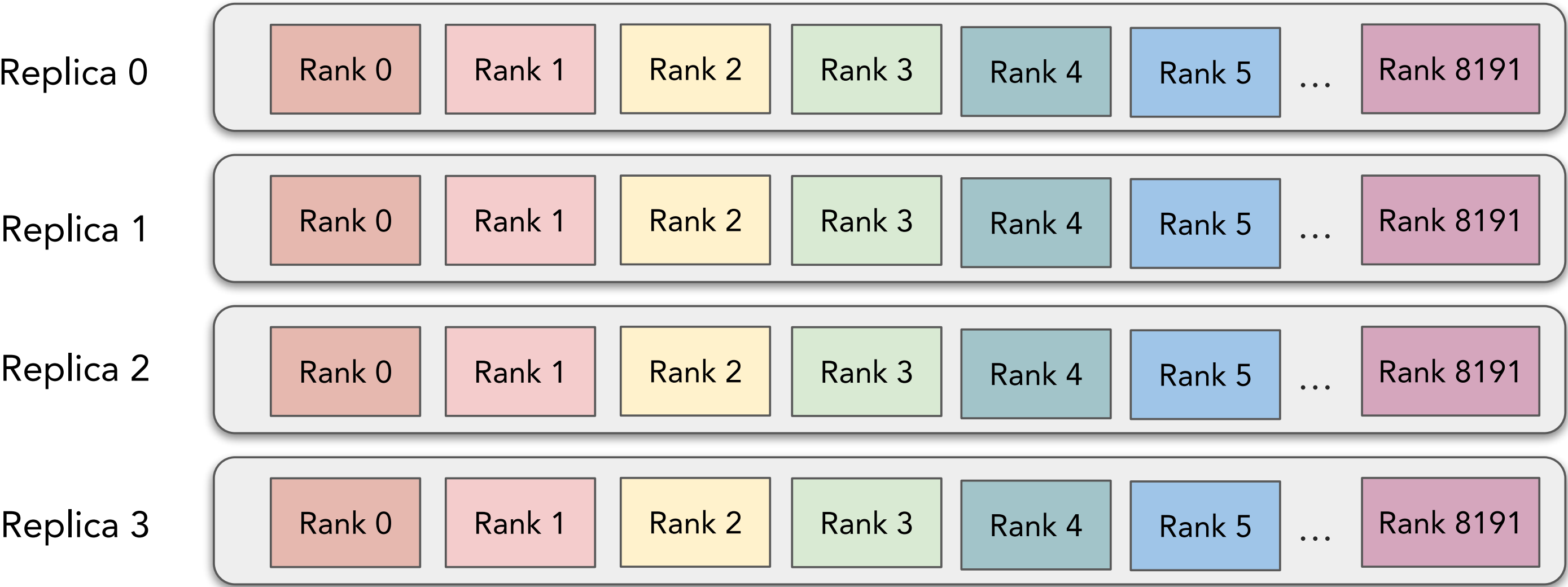
Old parallelism setup

We are training huge models...

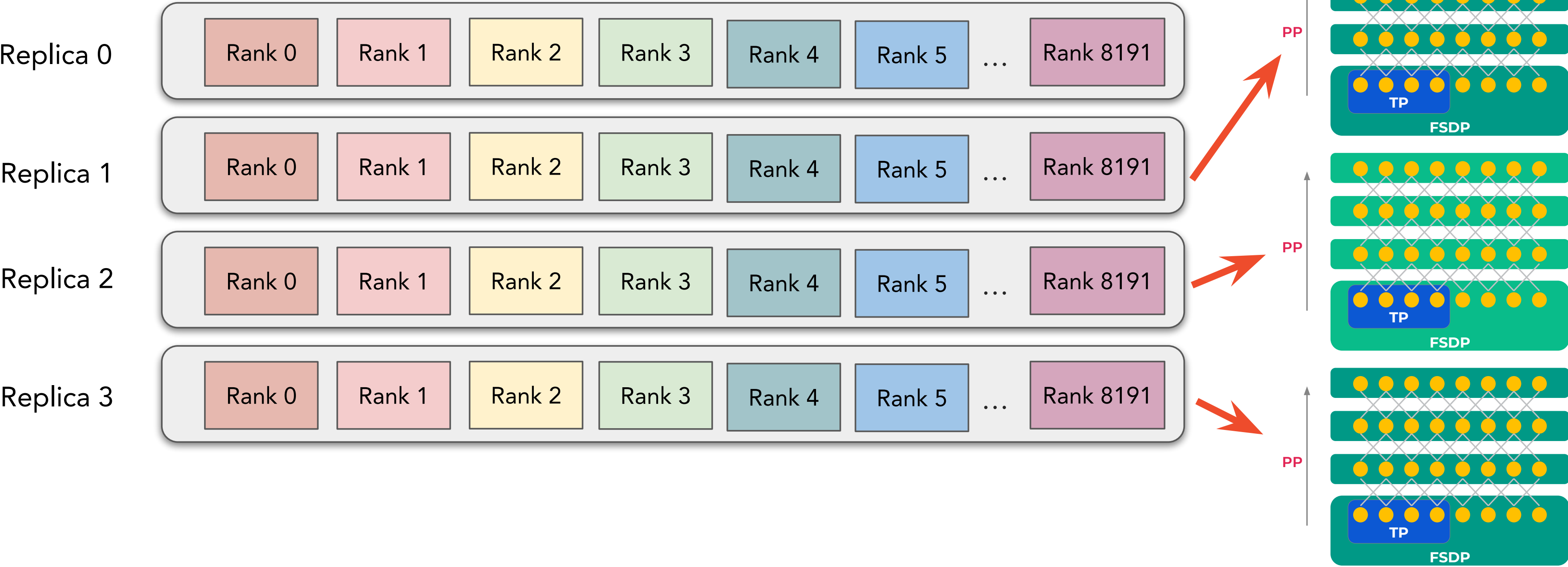
But each model doesn't need 100k GPUs.



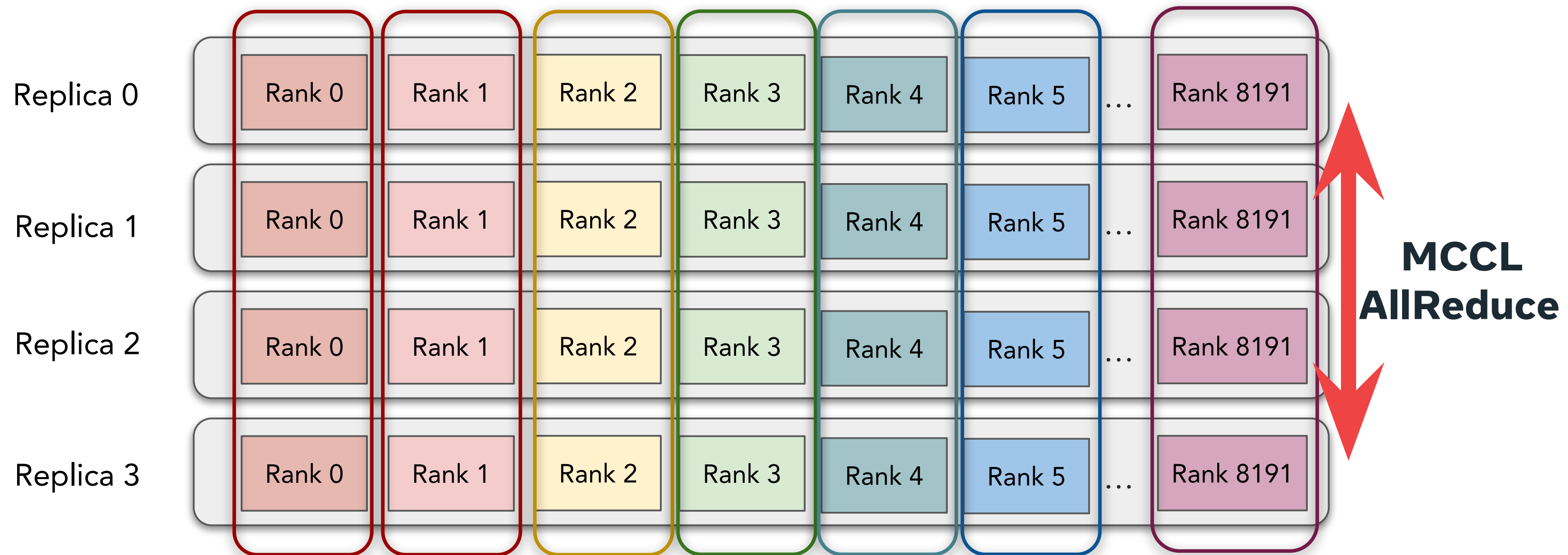
Scale up by adding replication



Scale up by adding replication

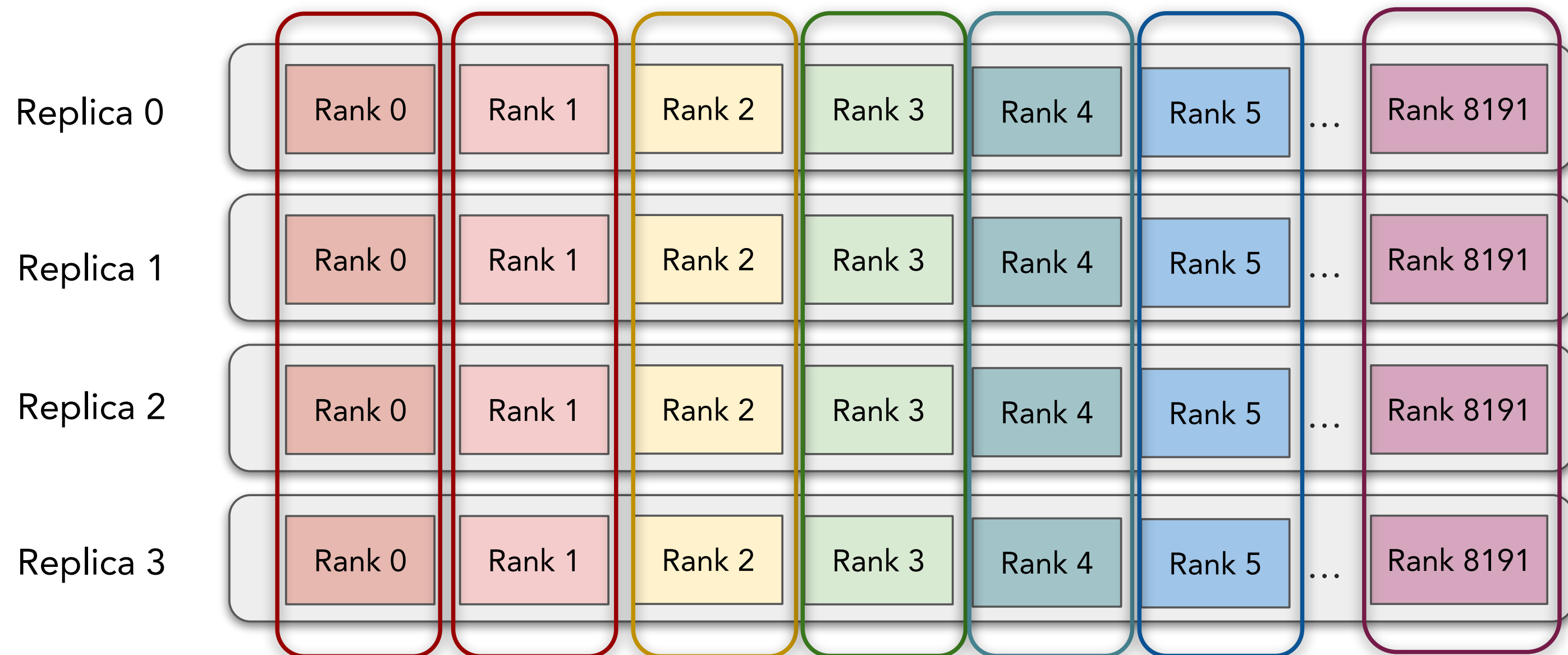


Using MCCL in large-scale distributed training



Gradient Synchronization via independent MCCL AllReduce's across matching ranks across replicas

Using MCCL in large-scale distributed training

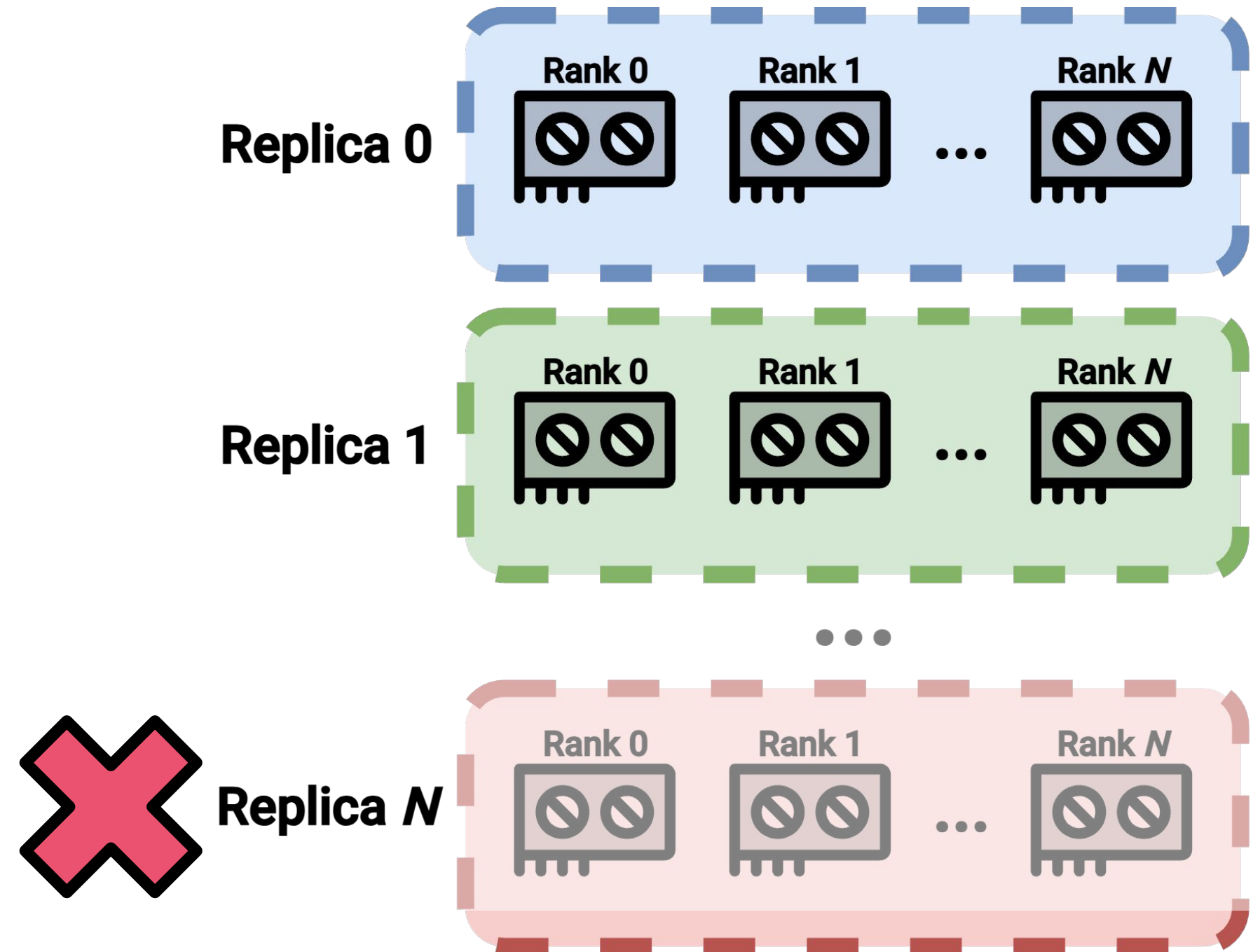


Model and optimizer state can be shared across different GPUs / ranks.

Replication setup available via existing PyTorch API: **HSDP (Hybrid-Sharded Data Parallelism)**

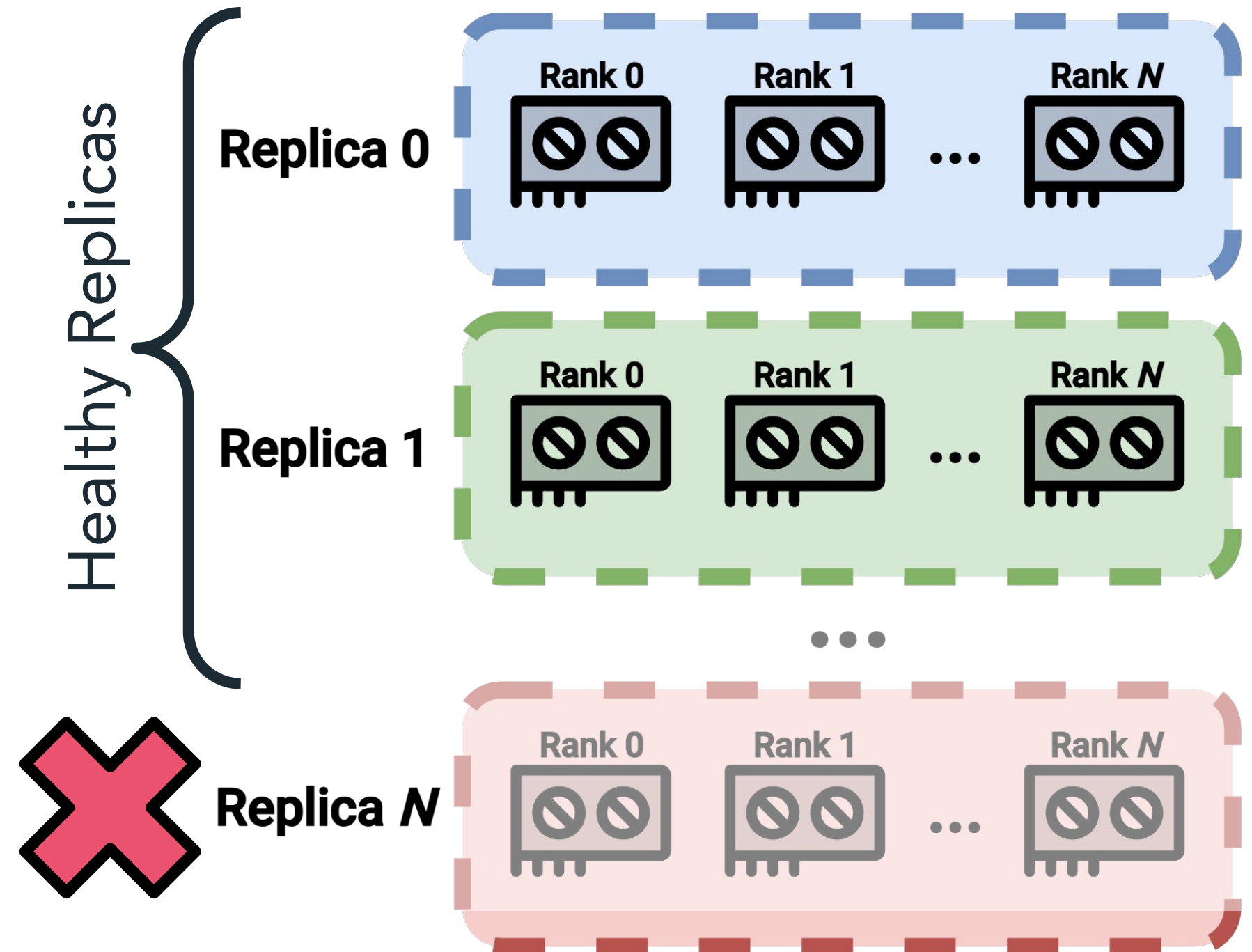
Replicas are the unit of fault isolation

- **Fail:** the affected replica drops out



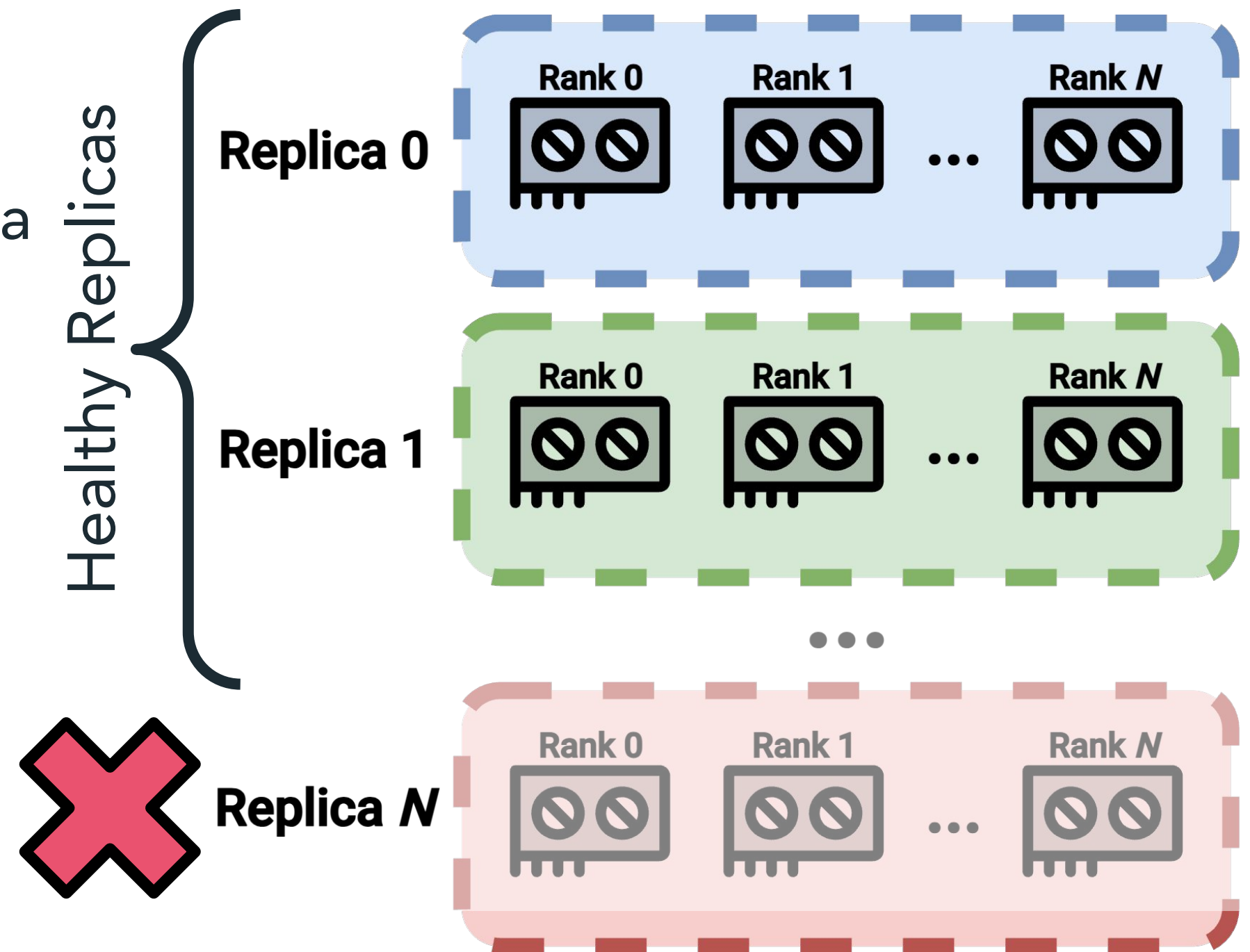
Replicas are the unit of fault isolation

- **Fail:** the affected replica drops out
- **Continue:** healthy replicas keep training



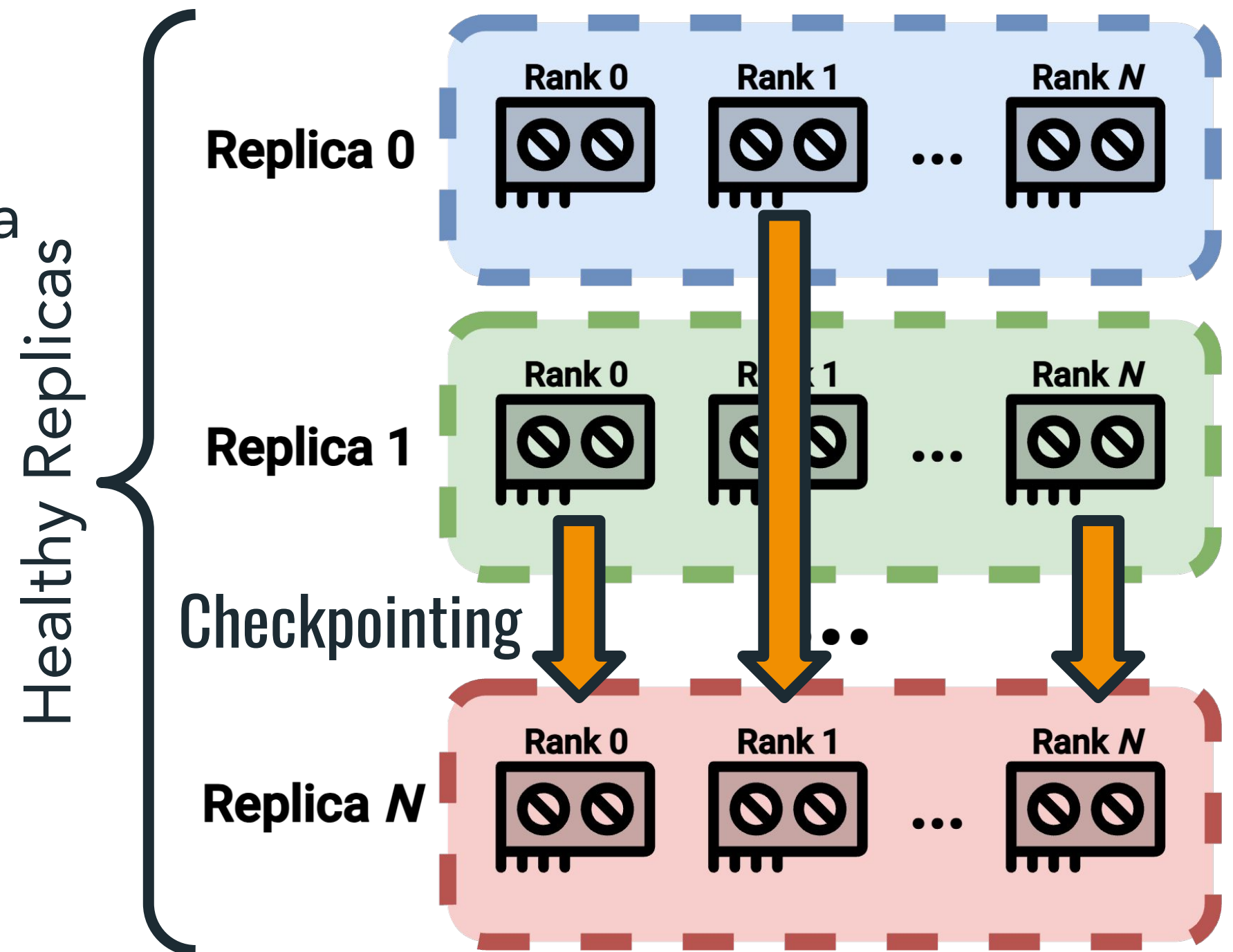
Replicas are the unit of fault isolation

- **Fail:** the affected replica drops out
- **Continue:** healthy replicas keep training
- **Reform:** new hardware rebuilds the replica



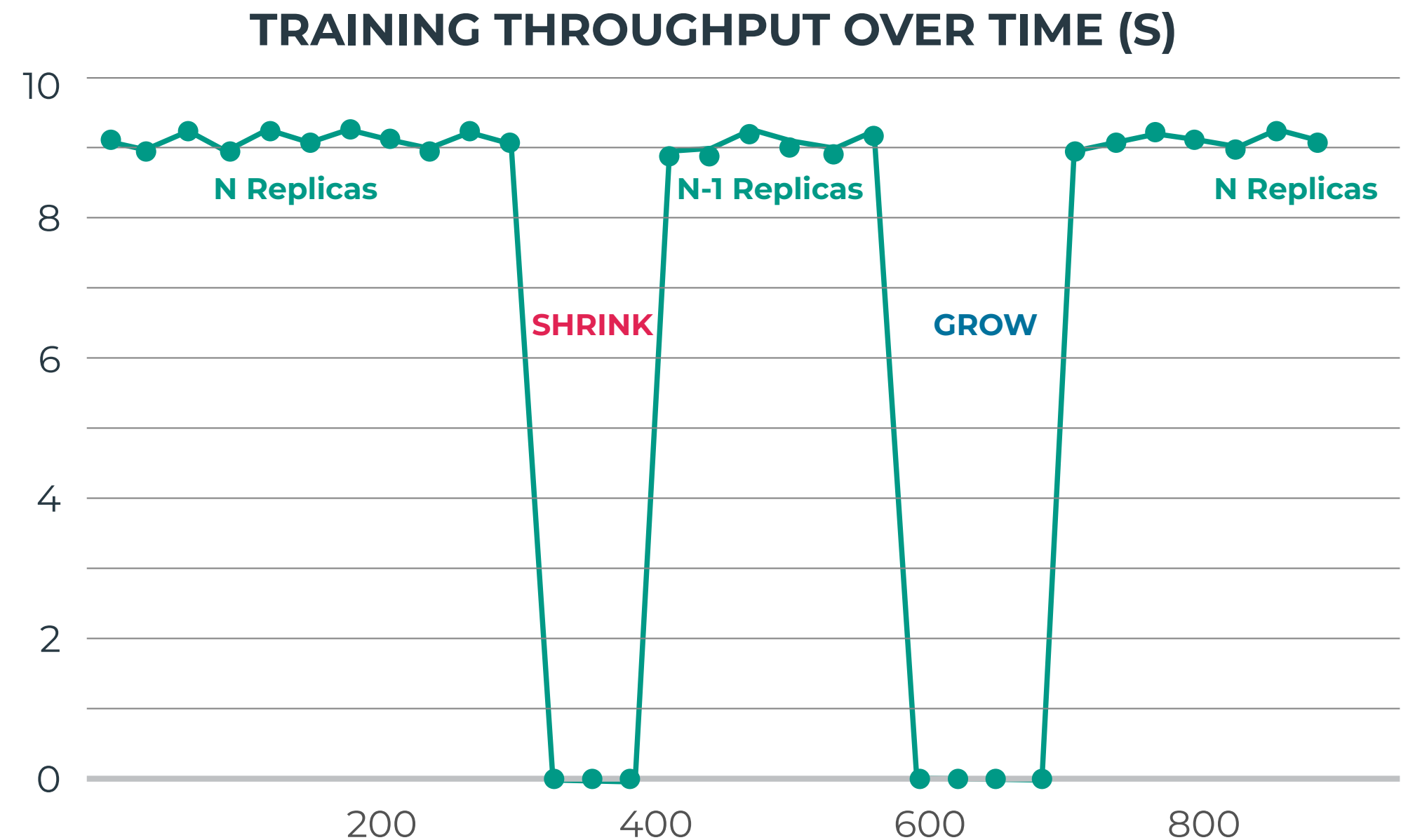
Replicas are the unit of fault isolation

- **Fail:** the affected replica drops out
- **Continue:** healthy replicas keep training
- **Reform:** new hardware rebuilds the replica
- **Rejoin:** it rejoins mid-run



Reconfigurable communicators

- **Shrink:** exclude a failed rank, keep everybody else running
 - Collective timeout
 - Quorum reformation
 - Retry training step



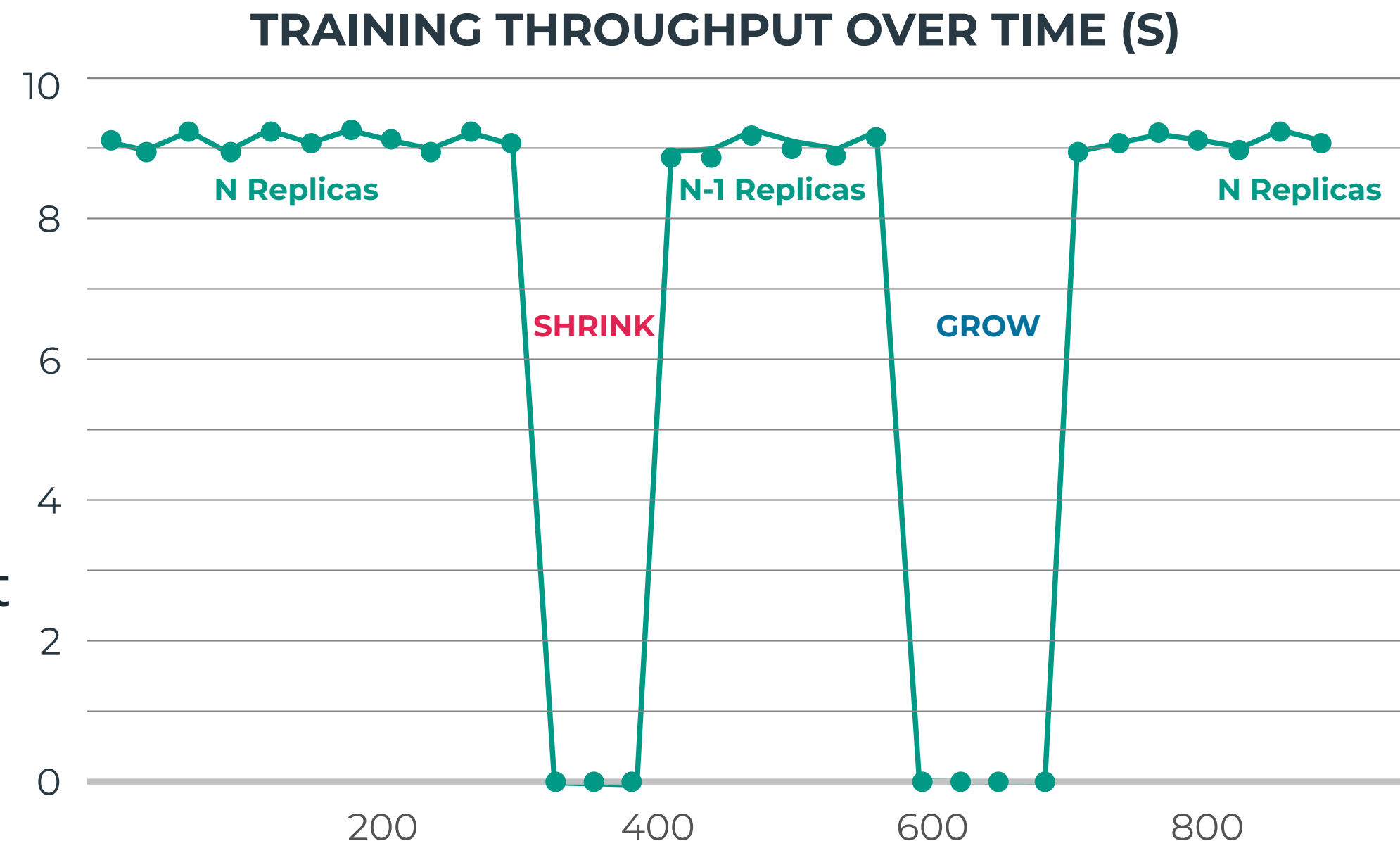
Reconfigurable communicators

- **Shrink:** exclude a failed rank, keep everybody else running

- Collective timeout
- Quorum reformation
- Retry training step

- **Grow:** admit replacement hardware

- Quorum reformation
- Load model/optimizer checkpoint
- Load data checkpoint
- Continue training what replacement replica



Reconfigurable communicators

- **Shrink:** exclude a failed rank, keep everybody else running

- Collective timeout

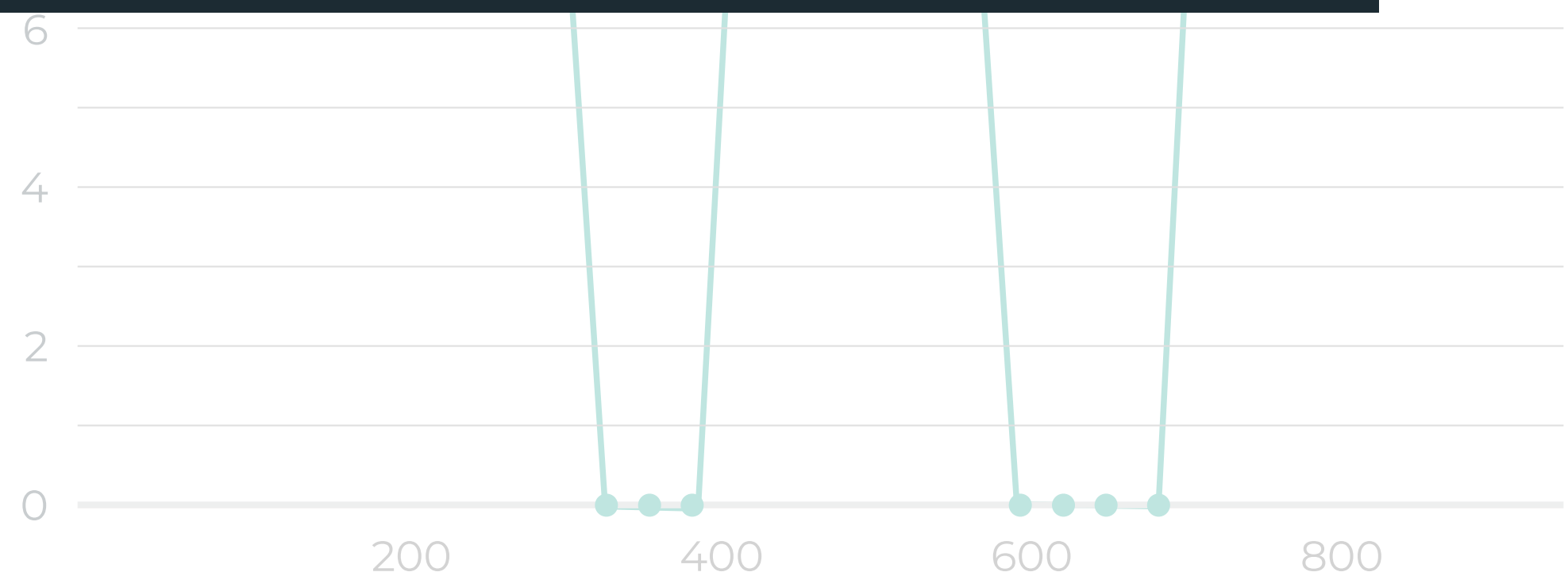
- Communicator failure

-

-

- **Grow:**
-
- Load model/optimizer checkpoint
- Load data checkpoint

Recovery and replacement are decoupled



Cooperative abort API

Timeout-based

Set a per-collective timeout for automatic detection and abort

User-driven

Explicitly call `abort()` on receiving an external health signal

Non-Blocking Propagation

Propagates asynchronously through the lower transport layers

Pending Handle Status

Pending operations report failure as abort is observed

Host (CPU)

Device (GPU)

Abort

Host-Level Controller

Triggered by timeout, user `abort()`, or bootstrap poll error

AbortState

Mapped Pinned Host Memory

Shared state across host & PCIe with first-writer-wins reason

AbortDevice

Device-Level Execution

Per-block copy on GPU; breaks wait loop for clean kernel exit

Cooperative abort API

Timeout-based

Set a per-collective timeout for automatic detection and abort

Abort

Host-Level Controller

Triggered by timeout, user `abort()`, or bootstrap poll error

Recovery builds on **abort**.
Unwind old communication, establish a **safe boundary**, **reform the quorum**.

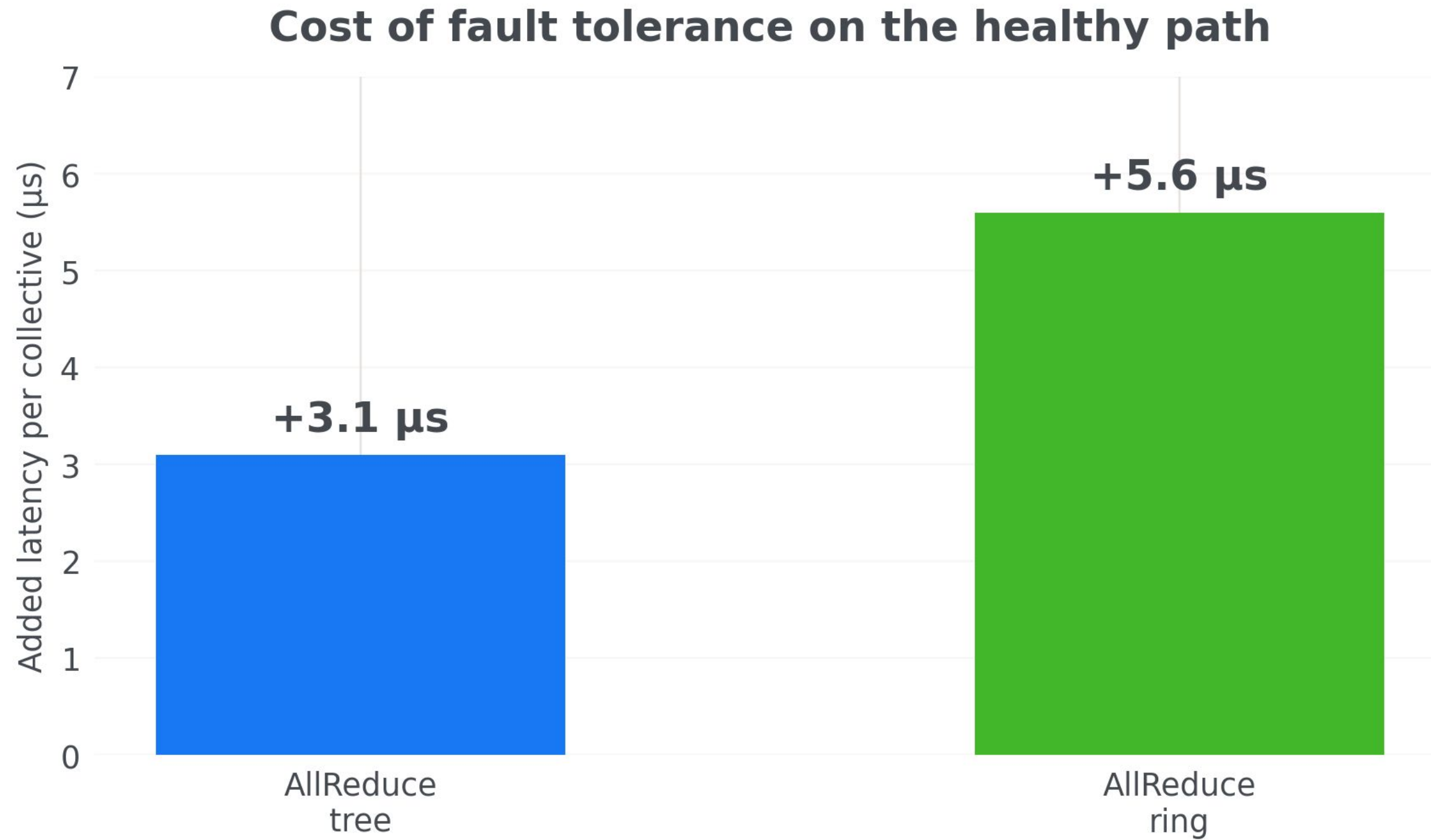
Pending Handle Status

Pending operations report failure as abort is observed

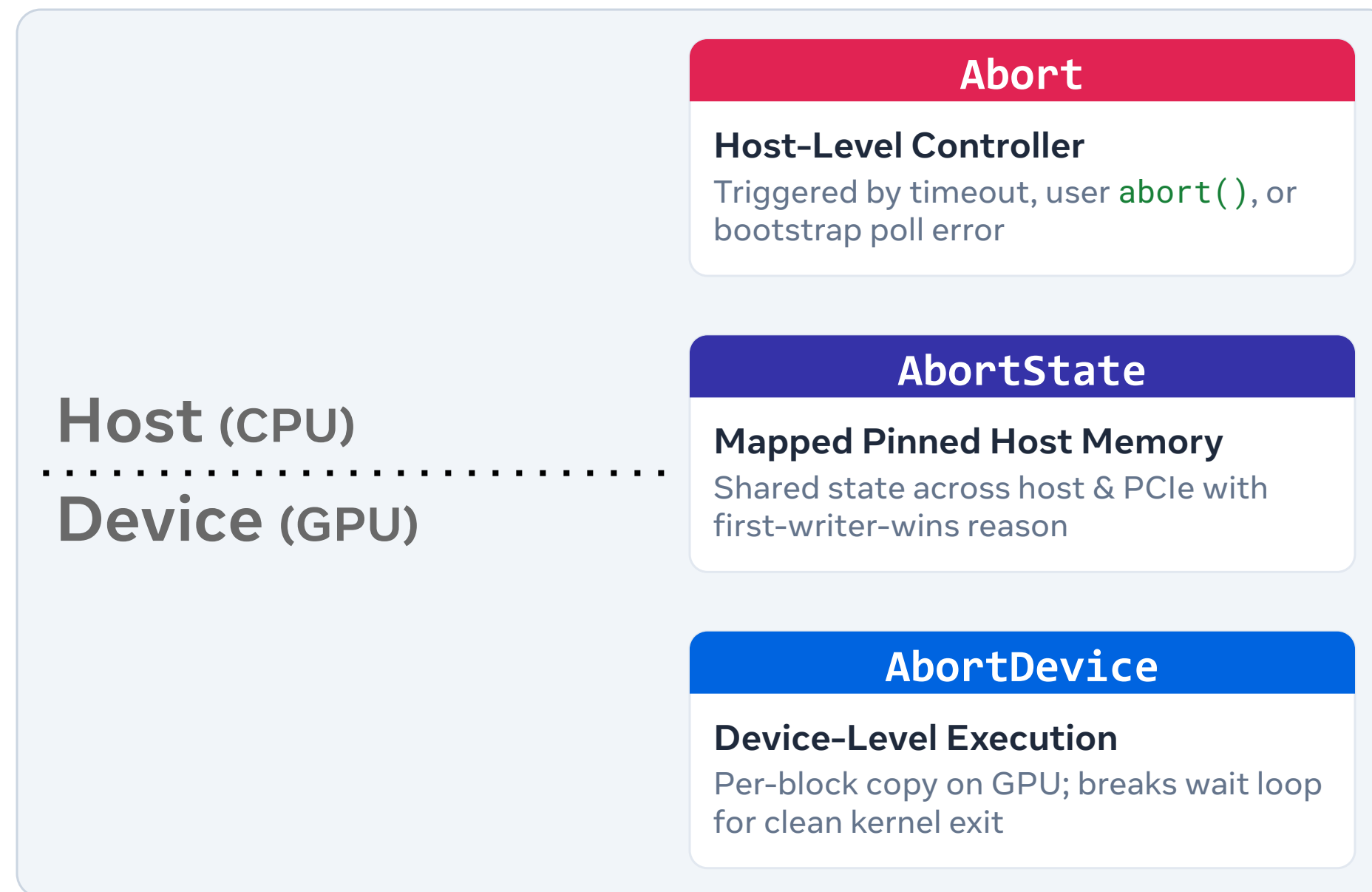
Device-Level Execution

Per-block copy on GPU; breaks wait loop for clean kernel exit

Low healthy-path fault tolerance overhead



Fault tolerance without a throughput tax



Keep **abort-deadline** checks GPU-local

Restrict **deadline/timeout checks** to the

Fault tolerance without a throughput tax

Abort

Host-Level Controller

Triggered by timeout, user `abort()`, or bootstrap poll error

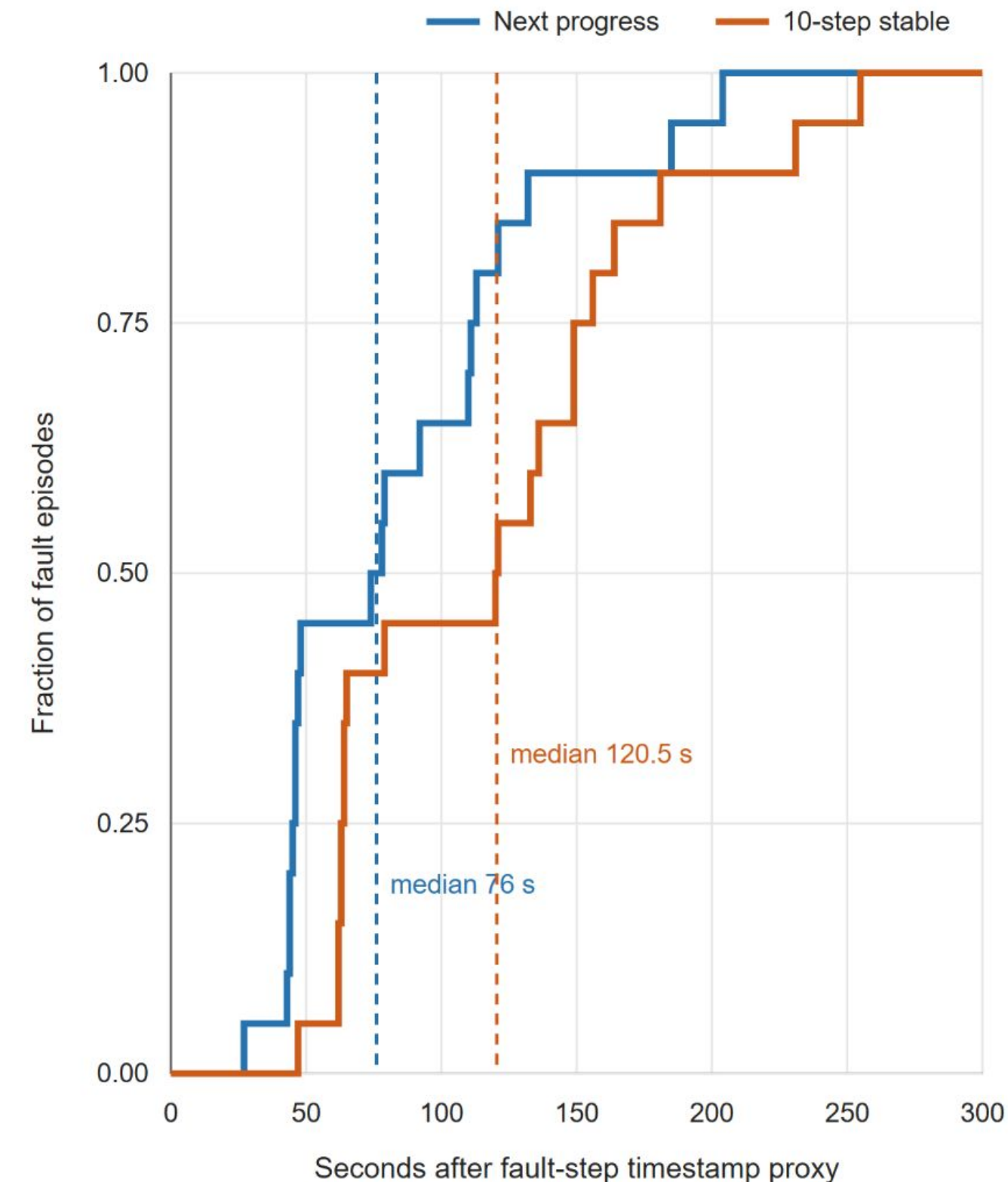
Small delay in seeing **abort** vs. **reduced overhead during healthy execution**

AbortDevice

Device-Level Execution

Per-block copy on GPU; breaks wait loop for clean kernel exit

Process recovery, increased training efficiency via MCCL

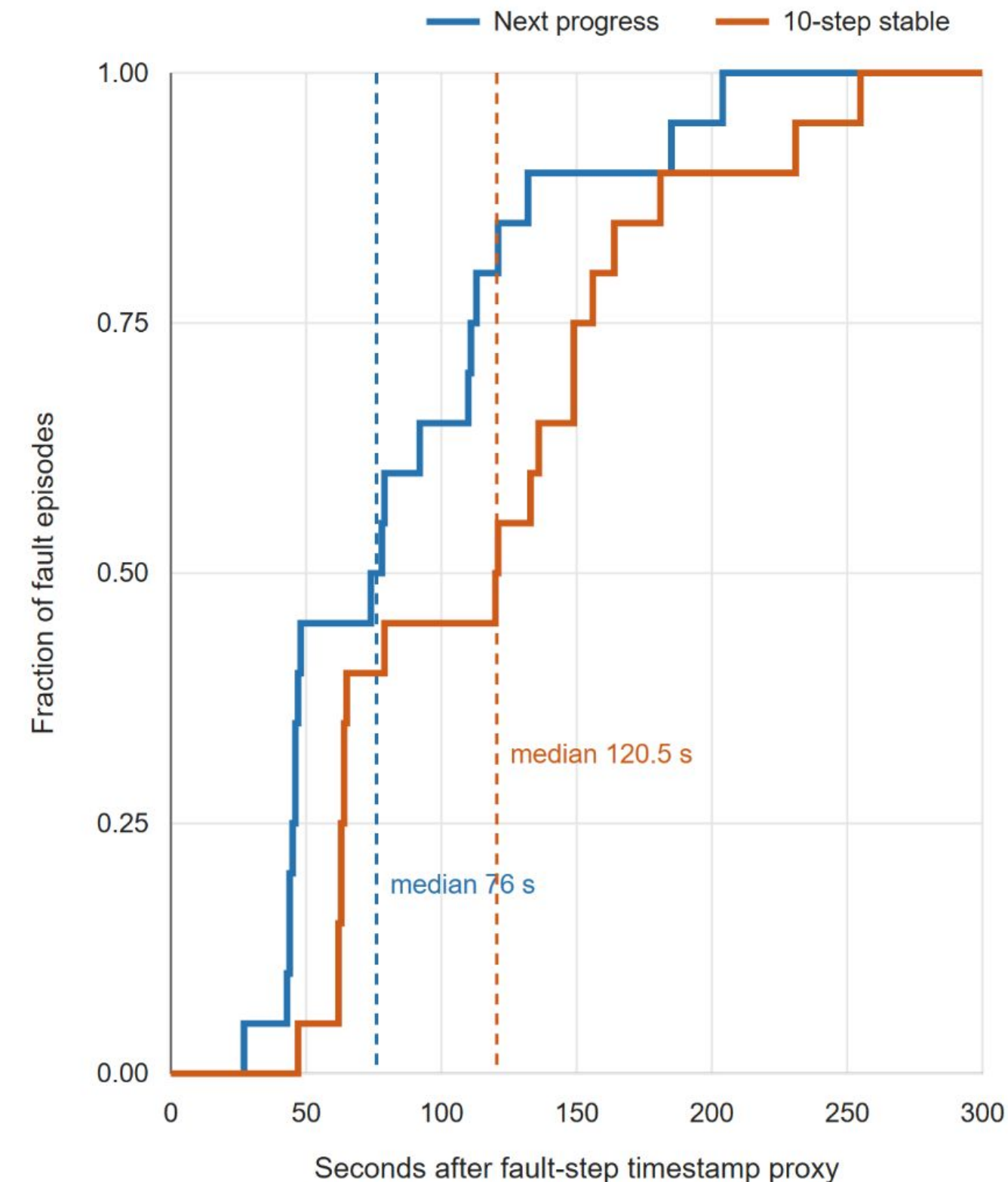


Training “**goodput**”: rate of **useful training progress**

Median time-to-next-progress: ~76 seconds

Median time-to-pre-fault-throughput: ~2

Process recovery, increased training efficiency via MCCL



Training “**goodput**”: rate of useful training progress

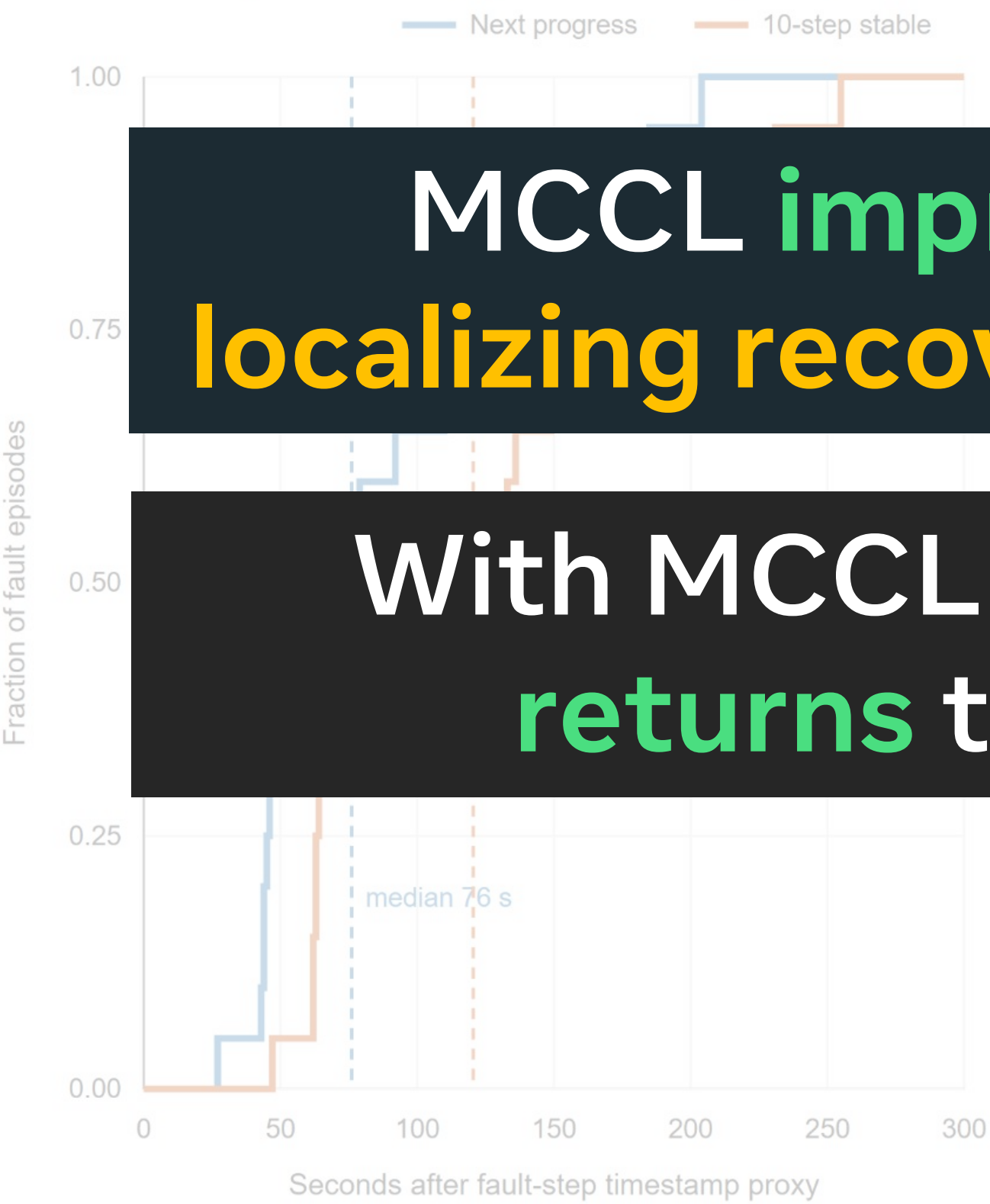
Median time-to-next-progress: ~76 seconds

Median time-to-pre-fault-throughput: ~2

MCCL optimizes **shrink** via **FastAbort** mechanism.

MCCL optimizes **grow** via **FT-RDMA send/recv**.

Process recovery, increased training efficiency via MCCL



MCCL improves training efficiency by localizing recovery and shortening downtime.

Median time-to-next-progress: ~76 seconds

With MCCL, the training system quickly returns to stable, useful progress.

MCCL optimizes grow via FT-RDMA send/recv.

```
import torch.distributed as dist

# Note: CUDA device, rendezvous, and MCCL tensor allocation setup omitted.

# 1. Establish initial membership
dist.init_process_group(
    backend="mccl", enable_reconfigure=True, timeout=TIMEOUT
)
handles = exchange_out_of_band(dist._get_reconfigure_handle())
dist._reconfigure(
    uuid=1000, handles=handles, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1000"},
).wait()

# 2. Run standard collectives
dist.all_reduce(tensor)

# 3. Only survivors execute this, after agreeing on dead_rank
survivors = [h for r, h in enumerate(handles) if r != dead_rank]
dist._reconfigure(
    uuid=1001, handles=survivors, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1001"},
).wait()

# 4. Resume collectives with renumbered ranks
dist.all_reduce(tensor)
```

```
import torch.distributed as dist

# Note: CUDA device, rendezvous, and MCCL tensor allocation setup omitted.

# 1. Establish initial membership
dist.init_process_group(
    backend="mccl", enable_reconfigure=True, timeout=TIMEOUT
)
handles = exchange_out_of_band(dist._get_reconfigure_handle())
dist._reconfigure(
    uuid=1000, handles=handles, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1000"},
).wait()

# 2. Run standard collectives
dist.all_reduce(tensor)

# 3. Only survivors execute this, after agreeing on dead_rank
survivors = [h for r, h in enumerate(handles) if r != dead_rank]
dist._reconfigure(
    uuid=1001, handles=survivors, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1001"},
).wait()

# 4. Resume collectives with renumbered ranks
dist.all_reduce(tensor)
```

```
import torch.distributed as dist

# Note: CUDA device, rendezvous, and MCCL tensor allocation setup omitted.

# 1. Establish initial membership
dist.init_process_group(
    backend="mccl", enable_reconfigure=True, timeout=TIMEOUT
)
handles = exchange_out_of_band(dist._get_reconfigure_handle())
dist._reconfigure(
    uuid=1000, handles=handles, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1000"},
).wait()

# 2. Run standard collectives
dist.all_reduce(tensor)

# 3. Only survivors execute this, after agreeing on dead_rank
survivors = [h for r, h in enumerate(handles) if r != dead_rank]
dist._reconfigure(
    uuid=1001, handles=survivors, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1001"},
).wait()

# 4. Resume collectives with renumbered ranks
dist.all_reduce(tensor)
```

```
import torch.distributed as dist

# Note: CUDA device, rendezvous, and MCCL tensor allocation setup omitted.

# 1. Establish initial membership
dist.init_process_group(
    backend="mccl", enable_reconfigure=True, timeout=TIMEOUT
)
handles = exchange_out_of_band(dist._get_reconfigure_handle())
dist._reconfigure(
    uuid=1000, handles=handles, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1000"},
).wait()

# 2. Run standard collectives
→ dist.all_reduce(tensor)

# 3. Only survivors execute this, after agreeing on dead_rank
survivors = [h for r, h in enumerate(handles) if r != dead_rank]
dist._reconfigure(
    uuid=1001, handles=survivors, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1001"},
).wait()

# 4. Resume collectives with renumbered ranks
dist.all_reduce(tensor)
```

```
import torch.distributed as dist

# Note: CUDA device, rendezvous, and MCCL tensor allocation setup omitted.

# 1. Establish initial membership
dist.init_process_group(
    backend="mccl", enable_reconfigure=True, timeout=TIMEOUT
)
handles = exchange_out_of_band(dist._get_reconfigure_handle())
dist._reconfigure(
    uuid=1000, handles=handles, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1000"},
).wait()

# 2. Run standard collectives
dist.all_reduce(tensor)

# 3. Only survivors execute this, after agreeing on dead_rank
survivors = [h for r, h in enumerate(handles) if r != dead_rank]
→ dist._reconfigure(
    uuid=1001, handles=survivors, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1001"},
).wait()

# 4. Resume collectives with renumbered ranks
dist.all_reduce(tensor)
```

```
import torch.distributed as dist

# Note: CUDA device, rendezvous, and MCCL tensor allocation setup omitted.

# 1. Establish initial membership
dist.init_process_group(
    backend="mccl", enable_reconfigure=True, timeout=TIMEOUT
)
handles = exchange_out_of_band(dist._get_reconfigure_handle())
dist._reconfigure(
    uuid=1000, handles=handles, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1000"},
).wait()

# 2. Run standard collectives
dist.all_reduce(tensor)

# 3. Only survivors execute this, after agreeing on dead_rank
survivors = [h for r, h in enumerate(handles) if r != dead_rank]
dist._reconfigure(
    uuid=1001, handles=survivors, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1001"},
).wait()

# 4. Resume collectives with renumbered ranks
dist.all_reduce(tensor)
```

```
import torch.distributed as dist

# Note: CUDA device, rendezvous, and MCCL tensor allocation setup omitted.

# 1. Establish initial membership
dist.init_process_group(
    backend="mccl", enable_reconfigure=True, timeout=TIMEOUT
)
handles = exchange_out_of_band(dist._get_reconfigure_handle())
dist._reconfigure(
    uuid=1000, handles=handles, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1000"},
).wait()

# 2. Run standard collectives
dist.all_reduce(tensor)

# 3. Only survivors execute this, after agreeing on dead_rank
survivors = [h for r, h in enumerate(handles) if r != dead_rank]
dist._reconfigure(
    uuid=1001, handles=survivors, timeout=TIMEOUT,
    hints={"reconfigureQuorumId": "1001"},
).wait()

# 4. Resume collectives with renumbered ranks
 dist.all_reduce(tensor)
```

What's next



01

Cross-Region Fault Tolerance

Expanding fault domain resilience across geographically distributed datacenters, handling inter-region latency and network partitioning.

Evolve our collective algorithms

Minimize bytes crossing regional boundaries

Aggregate locally, reduce latency-sensitive communication

What's next



Cross-Region Fault Tolerance

Expanding fault domain resilience across geographically distributed datacenters, handling inter-region latency and network partitioning.



Scaling to Many More Replicas

Supporting ultra-large scale cluster growth with low-overhead dynamic reconfigurations and efficient state synchronization.

What's next



01

Cross-Region Fault Tolerance

Expanding fault domain resilience across geographically distributed datacenters, handling inter-region latency and network partitioning.



02

Scaling to Many More Replicas

Supporting ultra-large scale cluster growth with low-overhead dynamic reconfigurations and efficient state synchronization.



03

More FT Collectives

Extending fault tolerant semantics to complex collective communication primitives and custom graph topologies.

Conclusion

- As training is scaling across larger and GPU clusters, **hardware failures** are **expected**.
- **MCCL**: low-overhead, fault-tolerant GPU communication.
 - **Shrink** preserves progress; **grow** restores capacity.
- Planned availability via PyTorch Distributed (**c10d**).
- **Validated at-scale** during training of Meta's frontier models, including **MUSE WONDER**.

Thanks for listening! Questions?



Speaker: Ben Carver

benrcarver@gmail.com · benrcarver@meta.com