

Already in use: [vLLM](#) · [SGLang](#) · [TensorRT-LLM](#) · [Dynamo](#) · [llm-d](#) · [veRL](#)

Fast transfers between elastic, named endpoints

NIXL (NVIDIA Inference Xfer Library) moves KV-cache blocks and model weights.
A single transfer API across GPUs, CPUs and storage. Your framework chooses the peers.

Fast hardware paths

NVLink within a fabric. InfiniBand / RoCE
across nodes. Direct GPU transfers.

Fault tolerance for resilient services

Peer failures surface as transfer errors.
Unrelated transfers can continue.
The framework decides how to recover.

Elastic endpoints: named agents

Connect peers by name, without fixed global ranks.
Add or replace workers, including Kubernetes pods.
Reconnect with fresh metadata and memory ranges.

Single API across memory tiers

Asynchronous READ / WRITE across GPU, CPU
and storage. Batch scattered blocks and reuse
registrations and prepared lists for very low latency.

KV cache: prefill computes, decode reuses

Prefill builds attention keys and values. Decode reuses them to generate tokens.

FRAMEWORK: PEER SELECTION, METADATA, LAYOUT AND READINESS



One endpoint initiates. Both sides prepare memory and agree on buffer lifetime. [1]

Bandwidth and latency: know the budget

NVLink 6 · Vera Rubin

3.0 TB/s

Per GPU, bidirectional; 1.5 TB/s one way. [4]

1 MiB payload floor: ≈0.70 μs

Bandwidth: hardware specifications. Time: bytes / peak one-way rate, before startup and overheads.

InfiniBand / RoCE · ConnectX-9

Up to 800 Gb/s

Total per adapter: 100 GB/s nominal. [5]

1 MiB payload floor: ≈10.5 μs

Applications using NIXL

Vera Rubin / CMX · KV-cache tier [8]

Up to 5× tokens/s

Long-context inference vs. traditional storage.
NIXL + Dynamo + BlueField-4.

ModelExpress · fast worker startup [3]

8 min → 1 min 44 s

Full replica startup; weights + JIT caches <10 s.
DeepSeek-V4 Pro · vLLM TP=8 · 8 × B200
NVIDIA report · July 2026 · ConnectX-7

A READ in native NIXL Python

1 Receiver: register and describe

```
from nixl_api import nixl_agent
a = nixl_agent("destination")
# dst: preallocated receiver CUDA tensor
reg = a.register_memory(dst)
# Source sends out of band after registration:
# peer_md: connection + memory metadata
# src_ranges: [(address, bytes, GPU ID), ...]
peer = a.add_remote_agent(peer_md)
local = a.get_xfer_descs(dst)
remote = a.get_xfer_descs(src_ranges, "VRAM")
```

2 Receiver: submit and complete

```
h = a.initialize_xfer(
    "READ", local, remote, peer)
state = a.transfer(h)
while state == "PROC":
    state = a.check_xfer_state(h)
if state != "DONE":
    raise RuntimeError("Transfer failed")
a.release_xfer_handle(h)
```

Repeated blocks

`h = a.make_prepped_xfer("READ", L, [0, 2], R, [3, 1])`

L / R: receiver-prepared local / remote lists. CUDA ordering and recovery stay explicit. [1]

RFC: a PyTorch Core interface

Goal: easier adoption, less maintenance for KV-cache exchange, live weight updates, model loading, multimodal transfers and cache offload.
A thin substrate for tensor/object lifetime management and CUDA stream synchronization.
Feedback on your specific use cases is welcome!