

Agents as Distributed Processes: A Native PyTorch Design Pattern for Multi-Agent Orchestration

Amit Rustagi



Abstract

Most multi-agent systems layer message brokers, state stores, and schedulers on top of PyTorch, adding infrastructure, serialization overhead, and operational complexity beyond what tightly coupled agent workloads require.

This poster introduces a native PyTorch design pattern that treats AI agents as first-class distributed processes. Agents coordinate through tensor-native communication, using shared-memory tensors where co-located, Process Groups for membership, and RPC for point-to-point interactions. Embeddings, confidence scores, and intermediate activations remain GPU-resident, avoiding unnecessary serialization and CPU transfers.

We demonstrate the approach with a code-repair system where a Planner proposes patches, an Executor validates them, and a Verifier accepts, iterates, or escalates results. The pattern targets tightly coupled, GPU-intensive reasoning loops rather than long-running workflow automation.

Attendees will leave with a practical design pattern for building multi-agent systems using native PyTorch as a distributed runtime instead of middleware-based orchestration.

Methodology



Agents as Distributed Processes

Each agent runs as a dedicated PyTorch process with its own model(s), state, and GPU context.



Shared-Storage Tensor Exchange (Co-located)

Tensor storage is placed in shared memory and tensor handles are passed between processes, minimizing copies for co-located communication.



Process Groups (Membership & Collectives)

Provide synchronization, barriers, and collective operations among participating processes.



RPC (Point-to-Point Interactions)

Asynchronous request/response and control-plane messages between agents.



GPU-Resident Tensors (Where Supported)

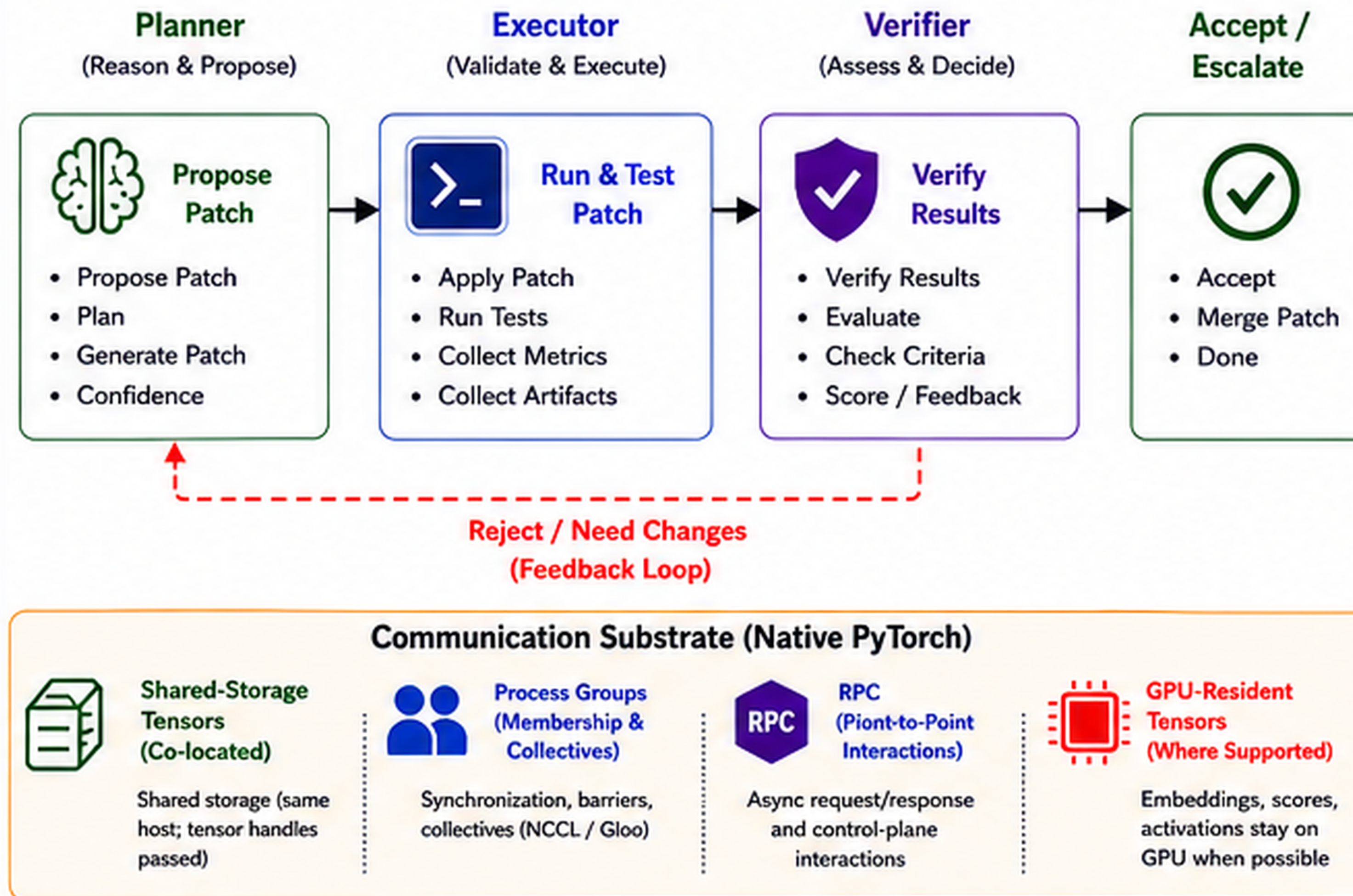
Embeddings, intermediate activations, and confidence scores stay on GPU: transfers occur only when necessary.



Iterative Coordination Pattern

Agents iteratively exchange tensors, evaluate, reflect, and terminate when acceptance criteria are met or escalation is required.

Architecture: Native PyTorch Multi-Agent Orchestration



All coordination is built with existing PyTorch distributed APIs: torch.distributed, torch.multiprocessing, and RPC.

Experiment: Code-Repair System



Setup

- 3 Agents (Planner, Executor, Verifier) as separate PyTorch processes
- Single node: 4x NVIDIA A100 80GB GPUs (PCIe)
- Models: Code LLM (Planner), Sandbox Runner (Executor), Verifier LLM (Verifier)
- Datasets: HumanEval, MBPP, and real-world bugs & snippets
- Identical prompts, temperature, and stop conditions across systems

Tools & Environment

- PyTorch 2.3, torch.distributed, torch.multiprocessing, torch RPC
- Python 3.11, CUDA 12.4, NCCL 2.20
- Linux (Ubuntu 22.04)
- 1 node, 4 x A100 80GB GPUs, 256GB RAM

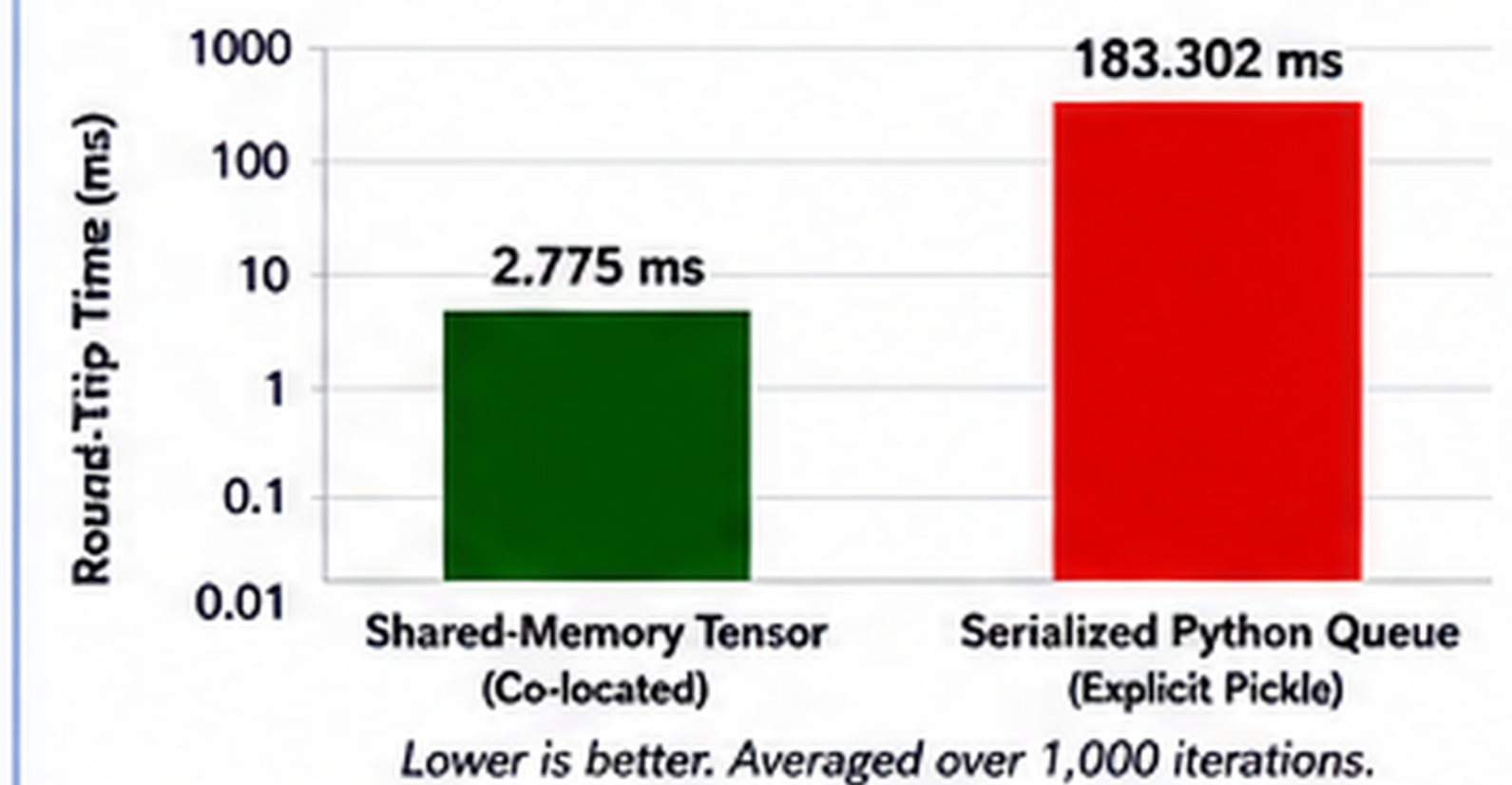


Panel A/B results are from our experiments under the setup above. Performance varies with hardware, workload, topology, and implementation.

Results

A. Communication Microbenchmark (Tensor Round Trip)

Single-node (A100 PCIe) — 32 MB tensor — Round trip time (ms)



66.1x
slower in
this benchmark

Baseline uses explicit pickle serialization via Python queue.

Metric	Baseline (Serialized Middleware*)	Native PyTorch Pattern	Improvement
End-to-End Success Rate	72.3%	83.6%	+11.3 pp
Average Iterations to Acceptance	238	162	-31.9%
End-to-End Latency (avg)	184.7 s	109.4 s	-40.8%
GPU Utilization (avg; mean across 4 GPUs)	61%	81%	+20 pp

* Baseline: Redis (message bus) + JSON/pickle + external scheduler.



Higher success rate is observed alongside lower coordination latency and fewer iterations to acceptance; causal attribution requires further evaluation.

C. Ablation: Communication Substrate Impact (32 MB tensor round trip)

Substrate	Serialization	Tensor Payload Handling	RTT / status
Shared-Storage Tensor (Co-located)	None	Shared storage (co-located CPU)	~2.8 ms
Process Group / NCCL (Collective)	None	Direct GPU tensor collective	Not reported
RPC (Point-to-Point Interactions)	Backend dependent	Backend / device dependent	Not reported
gRPC + Protobuf	Yes	Serialized payload	Not reported
Serialized Python Queue (Explicit Pickle)	Yes	Serialized payload	~180+ ms

Panel A/B values are experimental results from the setup above.

Panel C does not claim unreported RTT measurements for NCCL, RPC, or gRPC.



Treating tightly coupled agents as distributed PyTorch processes enables tensor-native coordination with low latency for the fast path.



Shared-storage tensors, Process Groups, and RPC for point-to-point interactions provide a substrate for high-throughput multi-agent reasoning loops.



Our communication microbenchmark shows a 66.1x speedup in round-trip time versus a serialized Python queue baseline, and we obtain meaningful gains on a realistic code-repair task (N = 120).



This pattern complements, not replaces, middleware-based systems for durability, workflow state, long-running orchestration, and operational tooling.



By using native PyTorch as the distributed runtime, we reduce unnecessary serialization, improve GPU utilization, and simplify the fast-path architecture.

● = Data / Tensor Flow - - -> = Feedback / Control Flow