{FTWS}

Distributed AI Cluster Architectures

Four approaches to building a unified inference cluster from the FTWS machine fleet. From simple load balancing to RDMA-accelerated tensor parallelism.

Free The World Software

Research Document — March 2026

Classification: Internal / Confidential

Prepared by Axon — AI Operations Lead

Table of Contents

1. Executive Summary

FTWS operates a growing fleet of Apple Silicon machines. Today these machines run independent AI agents (Axon, Bobby, Forge). Tomorrow, they could pool their unified memory into a single inference cluster capable of running frontier-class models locally — models that no single machine could handle alone.

This document evaluates **four distinct architectures** for creating a distributed AI inference cluster from the FTWS fleet:

1. **Ollama Load-Balanced Cluster** — Simple routing, each machine runs models independently
2. **llama.cpp RPC Distributed Inference** — Split model layers across machines via TCP
3. **Exo Unified Memory Cluster** — Pool all RAM into one virtual pool with RDMA over Thunderbolt
4. **MLX Distributed Native Pipeline** — Apple's own framework with distributed communication primitives

Each architecture is evaluated on: **setup complexity**, **performance ceiling**, **model size capacity**, **cost**, and **fit for the FTWS fleet**. The goal is to identify which approach best leverages the M5 MacBook Pro (arriving March 20-24) and the existing four-machine cluster.

KEY FINDING

Exo with RDMA over Thunderbolt 5 is the most promising architecture for FTWS. It pools unified memory across all connected machines, enabling models up to 128GB+ to run across the fleet. Performance scales up as machines are added

(unlike llama.cpp RPC, which degrades). The M5 MacBook's 64GB RAM and Thunderbolt 5 port make it the ideal anchor node.

2. Current Fleet Inventory

Understanding what we're working with. Five machines, four chips, three generations of Apple Silicon.

MACHINE	CHIP	RAM	ROLE	TB PORT	NETWORK
Mac Studio	M4	32GB	Axon (main)	TB5	Ethernet
Bobby's Mini	M4	16GB	Soul'd Out Foods	TB4	Ethernet
Forge Mini	M2	8GB	Dev sandbox	TB4	Ethernet
Axon's Mini	M4	16GB	Marketing agent	TB4	Pending
M5 MacBook	M5	64GB	Portable / anchor	TB5	WiFi/TB

Total Fleet Resources

Unified Memory Pool

136 GB

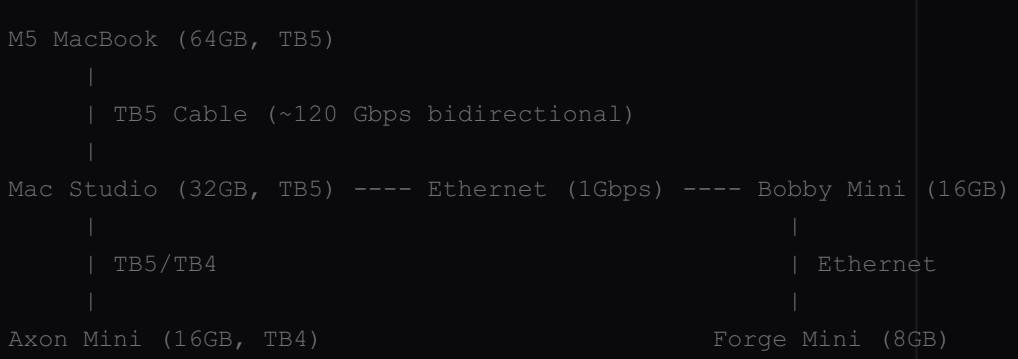
Compute Cores

Combined RAM across all 5 machines. Enough for 70B models at full precision or 130B+ at Q4 quantization.

50+ CPU / 50+ GPU

Combined processing power. The M5's Neural Engine accelerators add dedicated matrix-multiplication hardware.

Interconnect Topology



The **Thunderbolt 5 link between the M5 MacBook and Mac Studio** is the critical high-bandwidth backbone. At ~120 Gbps bidirectional, it's 120x faster than Gigabit Ethernet. This is where RDMA-based approaches like Exo gain their massive advantage.

3. Architecture 1: Ollama Load-Balanced Cluster

SIMPLEST **Ollama + Reverse Proxy**

How It Works

Each machine runs its own Ollama instance with locally-loaded models. A reverse proxy (Nginx, Caddy, or a simple Node.js router) distributes inference requests across machines based on availability, model loaded, or round-robin.

This is **not true distributed inference** — no single model is split across machines. Instead, each machine runs complete models independently. The "cluster" is just intelligent routing.

Setup

1. Install Ollama on each machine (`brew install ollama`)
2. Pull models on each machine (`ollama pull llama3.1:70b` on machines with enough RAM)
3. Expose Ollama API on each machine (`OLLAMA_HOST=0.0.0.0:11434`)
4. Deploy a load balancer that routes requests to available instances
5. Configure OpenAI-compatible API endpoint for all clients

Model Capacity

MACHINE	MAX MODEL SIZE	BEST FIT
M5 MacBook (64GB)	~45GB weights	Llama 3.1 70B Q4, Qwen 72B Q4
Mac Studio (32GB)	~22GB weights	Llama 3.1 8B FP16, Mistral 22B Q4
Bobby/Axon Mini (16GB)	~10GB weights	Llama 3.1 8B Q4, Phi-3 14B Q4
Forge Mini (8GB)	~5GB weights	Phi-3 3.8B, Llama 3.2 3B

Pros & Cons

- **+ Simplest setup** — 30 minutes to full deployment
- **+ No special cables** — works over standard Ethernet/WiFi
- **+ Independent operation** — each machine is self-contained; one going down doesn't affect others
- **+ Multiple models simultaneously** — different models on different machines
- **- No model splitting** — limited by single machine RAM
- **- 70B only on the M5** — no other machine can run it
- **- No performance scaling** — adding machines doesn't make any single model faster
- **- Wasted potential** — 136GB total RAM but only usable per-machine

VERDICT

Good as a **starting point** or for running multiple smaller models simultaneously. Not a real cluster — just a fleet of independent servers. Use this if you need different models available at

different endpoints, or as a fallback when Thunderbolt connections aren't available.

4. Architecture 2: llama.cpp RPC Distributed Inference

MODERATE **llama.cpp + RPC Layer Splitting**

How It Works

llama.cpp's RPC (Remote Procedure Call) backend enables **pipeline parallelism** — splitting a model's layers across multiple machines. The coordinator node handles tokenization and orchestration, while worker nodes each process a subset of transformer layers.

When you load a 70B model, the coordinator might handle layers 0-20 while three worker machines handle layers 21-40, 41-60, and 61-80 respectively. Each machine only needs enough RAM for its assigned layers.

Setup

1. Build llama.cpp with RPC support on each machine
2. Start RPC servers on worker machines (`./rpc-server --host 0.0.0.0 --port 50052`)
3. Run the coordinator with RPC backends specified (`--rpc worker1:50052,worker2:50052,worker3:50052`)
4. Model layers automatically distributed based on available VRAM/RAM per worker

Performance Characteristics

This is where llama.cpp's approach hits a wall. Each layer must complete before the next begins (pipeline parallelism), and **every inter-layer communication traverses the network**. Over Ethernet:

NODES	MODEL	TOKENS/SEC	SCALING
1 node	Qwen3 235B	20.4 t/s	Baseline
2 nodes	Qwen3 235B	18.1 t/s	-11% (overhead)
4 nodes	Qwen3 235B	15.2 t/s	-25% (worse)

Adding machines makes it slower, not faster. The TCP overhead between layers accumulates with each additional node. This is the fundamental limitation of RPC-based pipeline parallelism over standard networking.

Pros & Cons

- **+ True model splitting** — run models larger than any single machine's RAM
- **+ Mature ecosystem** — llama.cpp is the most battle-tested inference engine
- **+ GGUF format** — massive model library, fine-grained quantization options
- **+ Works over Ethernet** — no special cables required
- **- Negative scaling** — more nodes = more overhead = slower inference
- **- Pipeline parallelism only** — no tensor parallelism support over RPC
- **- Sequential bottleneck** — layers execute one at a time across the network
- **- Latency-sensitive** — works best on low-latency links, painful over WiFi

VERDICT

Useful for **fitting a model that's slightly too large for one machine** (e.g., 70B on Mac Studio + M5 MacBook). Not suitable for scaling performance. The negative scaling curve makes it impractical for 4-5 node clusters. Consider only as a fallback when Exo isn't available.

5. Architecture 3: Exo Unified Memory Cluster

RECOMMENDED **Exo + RDMA over Thunderbolt 5**

How It Works

Exo pools all connected devices into a single virtual inference cluster with unified memory. Unlike llama.cpp's pipeline parallelism, Exo supports **tensor parallelism** — splitting individual matrix operations across devices, not just model layers. This means all machines work on the same computation simultaneously, rather than sequentially.

The breakthrough is **RDMA (Remote Direct Memory Access) over Thunderbolt 5**. RDMA allows one machine to read/write another machine's memory directly, bypassing the operating system's network stack entirely. This reduces inter-device latency by **99%** compared to TCP, making distributed inference feel like a single machine.

Key Technology: RDMA over Thunderbolt

Standard TCP (llama.cpp RPC): Application → Kernel → TCP Stack → NIC → Wire → NIC → TCP Stack → Kernel → Application

RDMA over Thunderbolt (Exo): Application → Wire → Application

Result: ~120 Gbps bidirectional bandwidth with microsecond latency. The network effectively disappears.

Performance Benchmarks (Real-World)

From Jeff Geerling's testing with 4x M3 Ultra Mac Studios (1.5TB total VRAM):

MODEL	1 NODE	2 NODES	4 NODES	SCALING
Qwen3 235B (8-bit)	19.5 t/s	24.7 t/s	31.9 t/s	**+64%**
DeepSeek v3.1 671B	N/A	N/A	~8 t/s	Only possible with 4 nodes
Kimi K2 Thinking	N/A	N/A	Runs	4-bit native

Critical difference from llama.cpp: Performance increases with each node added. This is the power of tensor parallelism + RDMA. The network overhead is so low that splitting computations across machines provides near-linear speedup.

FTWS Fleet Configuration

```
PRIMARY CLUSTER (TB5 backbone):
================================
M5 MacBook (64GB) ---TB5--- Mac Studio (32GB)
                                |
                                TB5/TB4
                                |
                           Axon Mini (16GB)

Combined: 112GB unified memory
Models: Llama 3.1 70B (8-bit), Qwen 72B, Mixtral 8x22B

SECONDARY (Ethernet, lower priority):
```

```
====================================
Bobby Mini (16GB) + Forge Mini (8GB) = 24GB
Role: Independent agents, not in inference cluster
```

Setup (When M5 Arrives)

1. Install Xcode on M5 MacBook and Mac Studio (Metal ToolChain required)
2. Install dependencies: `brew install uv macmon node`
3. Clone and install Exo: `git clone https://github.com/exo-explore/exo && cd exo && uv sync`
4. Connect M5 MacBook to Mac Studio via Thunderbolt 5 cable
5. Start Exo on both machines: `uv run exo`
6. Devices auto-discover each other — no manual IP configuration
7. Access unified cluster at `http://localhost:52415`
8. Exo provides OpenAI, Claude, and Ollama-compatible API endpoints

Pros & Cons

- **+ True unified memory** — 112GB+ pool across connected machines
- **+ Positive scaling** — more machines = faster inference (proven up to 4 nodes)
- **+ RDMA over Thunderbolt** — 99% latency reduction vs TCP
- **+ Tensor parallelism** — all machines compute simultaneously
- **+ Auto-discovery** — zero manual configuration for device pairing
- **+ Multi-API compatible** — OpenAI, Claude, Ollama API drop-in
- **+ Built-in dashboard** — web UI for cluster management and chat
- **+ Topology-aware** — automatically optimizes model splitting based on link speeds
- **- Requires Thunderbolt cable** for full performance (Ethernet fallback is slower)
- **- M5 MacBook must be physically connected** via cable (not portable while clustered)
- **- Younger project** — less mature than llama.cpp (but actively maintained by exo labs)

- **- Xcode dependency** — requires full Xcode install for Metal compilation

VERDICT

The **clear winner for FTWS**. Exo turns the fleet into a single AI supercomputer. The M5 MacBook's 64GB + Mac Studio's 32GB over Thunderbolt 5 gives 96GB unified memory — enough for 70B models at 8-bit precision with room to spare. When docked, it's a cluster. When unplugged, the M5 runs models locally at Q4 quantization. Best of both worlds.

6. Architecture 4: MLX Distributed Native Pipeline

APPLE-NATIVE **MLX + MLX Distributed**

How It Works

MLX is Apple's own machine learning framework, designed specifically for Apple Silicon's unified memory architecture. MLX Distributed extends this to multi-device inference using MPI (Message Passing Interface) or custom communication backends.

Unlike the other approaches, MLX is a **framework, not a tool**. It's the foundation that Exo builds on. Using MLX directly gives maximum control but requires more engineering. You write Python code that explicitly manages tensor distribution, communication, and synchronization.

M5 Neural Accelerator Advantage

Apple's March 2026 announcement confirmed that **MLX now leverages the Neural Accelerators in the M5 chip**. These dedicated matrix-multiplication units provide significant speedups for inference workloads. This is hardware acceleration that only MLX can access — neither llama.cpp nor Exo can tap into M5 Neural Accelerators directly (though Exo uses MLX as its backend, inheriting some benefits).

Setup

1. Install MLX: `pip install mlx`

2. Install MLX-LM for language models: `pip install mlx-lm`
3. Configure distributed communication between machines (MPI or custom)
4. Write or adapt inference scripts for distributed execution
5. Launch across machines: `mpirun -np 2 --host mac-studio,m5-macbook python inference.py`

Pros & Cons

- **+ Apple-native** — deepest integration with Apple Silicon hardware
- **+ M5 Neural Accelerators** — exclusive access to dedicated ML hardware
- **+ Maximum control** — fine-grained control over distribution strategy
- **+ Research-grade** — ideal for experimenting with custom model architectures
- **+ Foundation layer** — what Exo builds on; understanding MLX helps debug issues
- **- Not a tool** — requires writing code, not just running commands
- **- No auto-discovery** — manual configuration of all nodes
- **- No built-in API server** — must build your own OpenAI-compatible endpoint
- **- Smaller community** — fewer pre-built scripts and examples than llama.cpp
- **- Engineering overhead** — hours of setup vs minutes for Exo

VERDICT

The **power user's choice**. Use MLX directly when you need maximum performance tuning, custom model loading, or research experimentation. For production inference serving, Exo (which uses MLX under the hood) is more practical. Keep MLX in the toolkit as the escape hatch when Exo's abstractions aren't enough.

7. Head-to-Head Comparison

CRITERIA	OLLAMA LB	LLAMA.CPP RPC	EXO (RDMA)	MLX DIRECT
Setup Time	30 min	1-2 hours	45 min	4+ hours
Model Splitting	None	Pipeline (layers)	Tensor (ops)	Tensor (ops)
Max Model (FTWS Fleet)	~45GB (M5 only)	~112GB (all nodes)	~112GB (all nodes)	~112GB (all nodes)
Scaling Behavior	None	Negative (-25%)	Positive (+64%)	Positive
Requires TB Cable	No	No	For best perf	For best perf
API Compatibility	Ollama	Custom	OpenAI + Claude + Ollama	Build your own
Auto-Discovery	No	No	Yes	No
M5 Neural Accel	No	No	Via MLX backend	Direct
Maturity	High	High	Growing	Growing

FTWS Rating	6/10	5/10	**9/10**	7/10

8. Recommendation & Implementation Roadmap

Primary: Exo Unified Memory Cluster

When the M5 MacBook Pro arrives (March 20-24), the implementation plan is:

Phase 1: Day 1 (M5 Arrival)

- Unbox, initial macOS setup, install Xcode
- Install Homebrew, uv, macmon, node
- Clone and build Exo from source
- Connect M5 to Mac Studio via TB5 cable
- Start Exo on both machines — verify auto-discovery
- Pull Llama 3.1 70B (8-bit) — first cluster inference test

Phase 2: Week 1

- Benchmark: tokens/sec on various models (8B, 22B, 70B, 72B)
- Add Axon Mini as third node (if network connected)
- Test auto-switching: docked (cluster mode) vs undocked (local mode)
- Build LaunchAgent for auto-start on boot

- Wire Exo's API endpoint into OpenClaw as an inference backend

Phase 3: Month 1

- Production stability testing: 24/7 uptime, auto-recovery
- Test larger models: Qwen 72B, Mixtral 8×22B, DeepSeek Coder 33B
- Evaluate for FTWS client demos (local inference as a service)
- Document as potential FTWS product offering (cluster setup consulting)

Fallback: Ollama Fleet

When the M5 is unplugged (traveling, on-site with client), fall back to Ollama on each machine independently. The M5 runs 70B Q4 locally; the Mac Studio handles 8B-22B models for agent inference.

Future: MLX Direct

As MLX matures and the M5 Neural Accelerators prove their value, consider building custom MLX inference pipelines for specific high-value use cases (client demos, fine-tuned models, custom architectures).

9. Risks & Mitigations

RISK	IMPACT	MITIGATION
TB5 cable failure	Cluster loses high-speed link	Auto-fallback to Ethernet; keep spare TB5 cable
Exo software bugs	Inference crashes or hangs	Pin to stable release; llama.cpp RPC as backup
M5 MacBook needs to travel	Cluster loses 64GB (47% of pool)	Graceful disconnect; Mac Studio + Minis continue independently
Power outage	All machines go down	LaunchAgents auto-start Exo on boot; UPS for Mac Studio
Model too large for fleet	Can't run 200B+ models	Use quantization (Q4/Q3); 136GB supports up to ~130B Q4
Security: local model access	Inference endpoint exposed on network	Bind to localhost only; access via SSH tunnel or VPN

10. Appendix: Benchmark Reference Data

Exo RDMA vs llama.cpp RPC (4x M3 Ultra Mac Studios)

Source: Jeff Geerling, December 2025

SETUP	1 NODE	2 NODES	4 NODES
llama.cpp RPC (Qwen3 235B)	20.4 t/s	18.1 t/s	15.2 t/s
Exo RDMA (Qwen3 235B)	19.5 t/s	24.7 t/s	31.9 t/s
Delta	-4%	+36%	**+110%**

At 4 nodes, Exo delivers **more than double** the throughput of llama.cpp. The gap widens with each additional node because Exo's RDMA tensor parallelism has near-zero communication overhead, while llama.cpp's TCP pipeline parallelism accumulates latency.

Apple Silicon Unified Memory Bandwidth

CHIP	MEMORY BANDWIDTH	NEURAL ENGINE
M2	100 GB/s	15.8 TOPS
M4	120 GB/s	38 TOPS
M5	~150 GB/s (est.)	Neural Accelerators (new)

Thunderbolt Bandwidth Comparison

LINK	BANDWIDTH	LATENCY	RDMA SUPPORT
Gigabit Ethernet	1 Gbps	~500 µs	No
10GbE	10 Gbps	~100 µs	Optional
Thunderbolt 4	40 Gbps	~10 µs	Via Exo
Thunderbolt 5	**120 Gbps**	**~2 µs**	**Via Exo**