TECHNICAL WHITEPAPER

Language models, disassembled and rebuilt across a mesh of machines.

An architecture and protocol reference for a distributed transformer-inference engine: registry, node mesh, coordinator, lazy layer loading, KV caching, and quantization — built from first principles, without an inference framework underneath it.

v1.0
DOCUMENT REVISION

July 2026
PUBLISHED

MIT
LICENSE

github.com/rileynz/Distributed-LLM
SOURCE

CONTENTS

Table of contents

01 — ABSTRACT

Abstract

Distributed LLM is a from-scratch inference engine that splits a transformer's layers across independent machines and runs a single coherent forward pass across all of them — no one machine ever holds the full model in memory. It replaces the usual assumption behind LLM serving — one large, capable machine — with a coordination layer that lets any number of ordinary machines behave like one.

The system consists of four components: a **registry** that tracks which nodes are online and which layers each owns, a set of **nodes** that each compute only their assigned slice of the model, a **coordinator** that drives requests through the resulting chain, and a **hot-reload path** that keeps every machine's active model in sync. On top of this, three subsystems — lazy layer loading, a distributed KV cache, and optional quantization — exist specifically to make weak, heterogeneous hardware viable as inference infrastructure.

MEMORY	BANDWIDTH	COMPUTE
~13%	**1 tensor**	**2×–4×**
of checkpoint weights read by a node owning 2 of 16 layers	crosses the wire per generated token, post-prefill	RAM reduction per node under int8 / int4

02 — THE CONSTRAINT

The constraint

Running a capable language model has conventionally meant renting or owning a single machine large enough to hold every one of that model's layers in memory at once. Most available compute — across homes, labs, and desks — was never shaped like that: it exists as many small, disconnected slices rather than one large one.

The gap is not compute in the abstract, it is compute in one place. A cluster of ordinary machines, none individually capable of running the full model, can still run it together if the model itself can be taken apart along its layers and reassembled across a network. That requires three things: a transformer split cleanly enough that any machine can own a contiguous slice of it, a wire protocol precise enough to move intermediate state between machines without losing numerical correctness, and a mesh that stays coherent as machines join, leave, or are swapped mid-operation.

03 — SYSTEM ARCHITECTURE

System architecture

Four components in a single coherent chain: a phonebook, a set of compute slices, a driver, and a sync mechanism.

01 **The registry — the network's phonebook**

Nodes announce on startup, either with a fixed layer range or as an unassigned "pool" node awaiting one. The coordinator resolves the current live chain before every request, so the mesh can reshape itself between requests without a restart.

02 **Each node computes only its slice**

A node reads just its assigned layers off disk rather than the full checkpoint, receives a tensor from the previous node in the chain, runs its layers, and forwards the result on.

03 **The coordinator drives the request**

It tokenizes the prompt, sends it into the chain, and streams logits back as they arrive. The prompt crosses the wire once — after that, only the newest token's hidden state travels, since every node holds its own local KV cache.

04 **Hot reload reaches the whole mesh**

Every node polls the registry for the active model and its own assignment, so a model switch reaches every machine within seconds of being triggered from any one of them.

04 — REQUEST LIFECYCLE

Request lifecycle

A request passes through the mesh in two distinct phases:

Prefill

The coordinator resolves the live chain from the registry, tokenizes the incoming prompt, and sends the full prompt tensor to the first node in the chain once. Each node runs its layers over the entire prompt, populates its local KV cache keyed to a coordinator-issued request id, and forwards its output to the next node.

Decode

For every subsequent token, only the newest token's hidden state crosses the wire — each node reuses its own cached keys and values rather than recomputing attention over the whole sequence. Logits stream back to the coordinator token by token until an end-of-sequence token, a max-token limit, or the model's context window is reached.

> **Correctness note —** caches are local to each node's process and are never transmitted; only hidden states cross the wire, keyed by a per-request id. A cache is freed the moment its generation ends, and an idle-timeout sweep clears anything left behind by a crashed client after 10 minutes. A hot model swap invalidates every in-flight cache immediately, since cached keys/values are meaningless once the underlying weights change.

05 — LAZY LAYER LOADING

Lazy layer loading

Loading the entire checkpoint and discarding unused layers defeats the purpose of spreading a large model across weak devices — a node responsible for 2 of a model's 32 layers would still need enough RAM to momentarily hold all 32. Nodes instead read only their own layers directly off disk.

This was verified directly rather than by inspection: a 16-layer / 730MB test checkpoint loaded in full materializes 730MB of real tensors; the same checkpoint loaded for a node owning 2 layers materializes roughly 94MB — about 13% of the total — with the remaining tensors genuinely never read off disk.

CONDITION	BEHAVIOUR
.safetensors checkpoint, GPT-2 / OPT / Llama family	`Lazy load — only assigned layers read`
Pre-safetensors checkpoint format	`Fallback — full load, then slice`
Architecture outside the three supported families	`Fallback — full load, then slice`
Quantization active on that layer range	`Fallback — full load, then slice`

A node's own log line indicates which path was taken for its load. Combining lazy loading with quantization is deliberately out of scope for the current version — see Known constraints.

06 — DISTRIBUTED KV CACHING

Distributed KV caching

Every node keeps its own local cache of keys and values for each in-flight request. The prompt is processed once during prefill; from that point forward, only the single newest token's hidden state travels between nodes rather than the whole growing sequence, eliminating both redundant network traffic and redundant attention recomputation over already-processed tokens.

Because positions are cached incrementally rather than recomputed each step, generation stops cleanly once a model's context window is reached, rather than silently sliding a window over the prompt.

```
# disable caching to compare against always-recompute behaviour
python coordinator/client.py "your prompt" --no-cache
```

07 — QUANTIZATION

Quantization

Any catalogue model flagged quantizable can load in int8 or int4 via bitsandbytes in place of full precision, at a cost of roughly 2× to 4× less RAM per node — with some quality cost at int4. Quantization requires a CUDA-capable GPU on whichever node loads that layer range; nodes without one fall back automatically to full precision and print a warning rather than failing. Because quantization is applied at load time from the same downloaded checkpoint, there is no separate download step for it.

```
# switch models with an explicit quantization level
python download_model.py --switch mistral-7b --quant int4
```

08 — REGISTRY API REFERENCE

Registry API reference

ENDPOINT	METHOD	DESCRIPTION
/announce	POST	Node registers host, port, label, and hardware specs; includes a layer range if assigned, or omits both to join as a pool node.
/heartbeat	POST	Node keepalive, sent every 5 seconds.
/chain	GET	Ordered list of live, assigned nodes; returns 503 if the layer assignments have gaps.
/status	GET	Every node — assigned and pool — with liveness, requests served, and hardware specs.
/network_specs	GET	Aggregated hardware specs across alive, assigned nodes; used by the model recommender.
/pool	GET	Unassigned nodes awaiting a layer range and model.
/assignments	GET / POST	Per-node layer ranges, keyed by label; nodes poll this to pick up a new or changed assignment.
/active_model	GET / POST	The network-wide active model; nodes poll this so a switch reaches every machine, not only the one that issued it.
/health	GET	Liveness check.

09 — DEPLOYMENT TOPOLOGY

Deployment topology

Nodes are LAN-native by default and require no router configuration on a shared local network; each node auto-detects its own LAN IP to advertise to the registry, with a manual override for machines with multiple network interfaces. For machines spread across separate networks, a mesh VPN such as Tailscale makes them behave as if on the same LAN. Tensor transport currently uses Python's pickle format, which is appropriate for a trusted local network — node ports are not designed to face the open internet.

Startup order

Registry, then nodes, then coordinator/client — each in its own process, on any machine reachable on the mesh.

Auto layer split

Nodes can join a "pool" with no fixed range; the model manager evenly re-splits a chosen model's layers across every currently confirmed node.

10 — KNOWN CONSTRAINTS

Known constraints

Centralized registry
A single point of coordination — if it drops, new requests cannot be routed and model switches cannot propagate, though in-flight requests still complete. A peer-to-peer topology would remove this but is a materially larger build.

No load balancing
Requests run through one linear chain; concurrent requests queue rather than route across spare capacity.

Node failure is detected, not routed around
A dropped node is detected within seconds and the request returns whatever it generated so far. It is not auto-retried — with KV caching in play, a silent resend risks double-applying a token.

Lazy loading and quantization are not yet composable
Quantized loads use the full-load-then-slice path; safely packing weights from hand-picked tensors is separate, riskier work.

Tensor transport uses pickle
Appropriate for a trusted network only. Node management ports are not hardened for exposure to the open internet.

11 — ROADMAP

Roadmap

- → Range-request partial downloads for single-file, non-sharded checkpoints
- → Prompt-prefix cache reuse across separate requests
- → Continuous batching in place of request queuing
- → Automatic rerouting around a dead node
- → CPU-friendly quantization path
- → Streaming HTTP API for external client applications

12 — WIRE PROTOCOL SPECIFICATION

Wire protocol specification

Nodes communicate over plain TCP sockets using a small length-prefixed framing format rather than an existing RPC or serving framework. Every message on the wire has the same two-part shape:

FIELD	SIZE	DESCRIPTION
`length header`	8 bytes	Unsigned 64-bit big-endian integer — the byte length of the payload that follows.
`payload`	variable	A pickled Python object — typically a dict containing a tensor, a request id, and routing metadata.

The receiving side reads exactly 8 bytes to learn the payload length, then reads exactly that many bytes before unpickling — there is no delimiter to search for and no ambiguity about where one message ends and the next begins, even over a stream socket that may deliver a message in several fragments.

```
# conceptually, every send is:
header = struct.pack(">Q", len(payload_bytes))
socket.sendall(header + payload_bytes)
```

Why pickle — it serializes PyTorch tensors and Python dicts with no custom format to design or maintain, which keeps the protocol small enough to read end to end. The tradeoff is explicit: pickle can execute arbitrary code on deserialization, so this framing is appropriate only between machines that already trust each other on a private network. It is not a protocol to expose to the open internet — a public-facing version would swap the payload format for JSON headers plus a safetensors-style raw tensor buffer, which carries no code-execution risk on unpickling.

13 — NODE PROCESS ANATOMY

Node process anatomy

A single node process exposes two separate surfaces rather than one: an inference socket that only ever speaks the wire protocol above, and a small HTTP management server used for control-plane traffic — hardware-spec reporting, hot-reload polling, and triggering a shard download. Keeping these separate means the hot path that moves tensors between nodes carries nothing but tensors.

Inference socket

Accepts a hidden-state tensor and a request id, runs the node's assigned layer range, and returns the result — either forwarding it on for a mid-chain node, or returning logits at the end of the chain.

Management HTTP server

Reports hardware specs to the registry, serves download-progress polling, and answers the hot-reload watcher's periodic checks against the registry's active-model and assignment endpoints.

Two background loops run alongside request handling: a **hot-reload watcher** that polls the registry every few seconds for a changed active model or a changed layer assignment and triggers a reload in place when either changes, and a **cache sweeper** that runs on a fixed interval and frees any per-request KV cache left behind by a client that crashed or disconnected mid-generation without a clean close.

14 — PROVISIONING & THE CONFIRMATION GATE

Provisioning & the confirmation gate

Before any model is downloaded or split, every call site — the CLI, the launcher's model menu, and the web dashboard — routes through one shared provisioning path rather than three separate copies of the same logic. That path enforces a hard-stop confirmation: it lists every node the registry currently considers alive, pool and already-assigned alike, and blocks until the operator explicitly confirms that list. Nothing is split or pushed before that confirmation, and a fingerprint of the confirmed node set is checked again at the moment of download so a stale confirmation can never green-light a different set of machines than the one actually reviewed.

Capacity-proportional splitting

Once confirmed, a model's layers are split across every node in the set — proportional to each node's free RAM rather than divided evenly, with an even split used as the fallback when hardware information isn't available. This lets a heterogeneous fleet — a 4GB laptop alongside an 8GB desktop — pool into what behaves like one larger virtual device, with the larger machine taking a proportionally larger slice instead of both being capped to the smaller one's share. Switching models re-runs this split fresh across the whole confirmed set every time, rather than only provisioning newly added nodes.

15 — BENCHMARKING & MODEL RECOMMENDATION

Benchmarking & model recommendation

Each node runs a short local timing pass — a matrix-multiply throughput proxy plus a real forward-pass timing on a small reference model — and reports the result to the registry over its regular heartbeat. These per-node scores are aggregated into a single network score, which the model manager uses to recommend catalogue entries the current mesh can run smoothly rather than merely load.

SCORE	TIER	WHAT FITS
0+	Minimal	DistilGPT-2 or GPT-2 Small only
10+	Light	GPT-2 Small / Medium, OPT-125M
25+	Standard	GPT-2 Medium / Large, OPT-350M, TinyLlama
50+	Performance	GPT-2 XL, OPT-1.3B, GPT-Neo 1.3B
100+	High	OPT-2.7B, GPT-Neo 2.7B and above

16 — MODEL CATALOGUE

Model catalogue

The catalogue spans three architecture families — GPT-2-style, Meta's OPT, and the Llama/RoPE family that also covers TinyLlama, Phi-3, Mistral, and Qwen2.5 — chosen because each family shares one internal module layout, so supporting a family's layout once is enough to support every model built on it.

MODEL	PARAMS	DOWNLOAD	MIN. NODES	QUALITY
DistilGPT-2	82M	0.35 GB	1	1 / 5
GPT-2 Small	124M	0.5 GB	1	2 / 5
GPT-2 Medium	345M	1.4 GB	2	3 / 5
GPT-2 Large	774M	3.0 GB	2	3 / 5
GPT-2 XL	1.5B	6.0 GB	2	4 / 5
GPT-Neo 1.3B	1.3B	5.0 GB	2	4 / 5
OPT 1.3B	1.3B	5.0 GB	2	4 / 5
TinyLlama 1.1B recommended	1.1B	2.2 GB	1	4 / 5
Phi-3 Mini	3.8B	7.6 GB	2	5 / 5
Mistral 7B Instruct	7B	14.5 GB	3	5 / 5

Qwen2.5 14B Instruct	14B	29.5 GB	3	5 / 5

TinyLlama 1.1B is the recommended entry point: a modern RoPE architecture trained on 3 trillion tokens, a materially better quality-to-size ratio than any GPT-2 or OPT variant, and it runs on a single node. Qwen2.5 14B sits at the other end deliberately — designed to be split across several modest machines via capacity-proportional provisioning rather than run on one large one.

17 — COMPARISON TO RELATED SYSTEMS

Comparison to related systems

Petals (BigScience) established the core idea this project builds on: split a large model's layers across volunteer machines and route requests through them as a chain. Distributed LLM follows the same shape at a smaller, more traceable scale.

PROPERTY	PETALS	DISTRIBUTED LLM
Topology	Peer-to-peer DHT	Centralized registry
Load balancing	Yes, across replicas	Not yet — single chain
Fault routing	Automatic rerouting	Detected, not rerouted
Codebase	Production-scale, complex	Small, readable end to end

The comparison is deliberate rather than competitive: Petals solves the single point of failure a central registry introduces, at the cost of a materially larger and harder-to-trace system. This project takes the opposite tradeoff on purpose, favoring a codebase where every piece — registry, node, coordinator, protocol — can be read start to finish in one sitting.

18 — GLOSSARY

Glossary

TERM	DEFINITION
Chain	The ordered sequence of assigned nodes a request passes through, resolved from the registry before each request.
Pool node	A node that has announced itself to the registry without a fixed layer range, awaiting assignment.
Prefill	The first phase of a request, where the full prompt is processed once and each node's KV cache is populated.
Decode	The token-by-token phase following prefill, where only the newest hidden state crosses the wire per step.
Lazy loading	Reading only a node's assigned layer tensors off a safetensors checkpoint, rather than loading the full checkpoint into memory first.
Hot reload	A model or assignment change propagating to every node in the mesh via registry polling, without a process restart.
Capacity-proportional split	Dividing a model's layers across confirmed nodes in proportion to each node's free RAM, rather than evenly.

19 — MEASURED RESULTS (REPRODUCED IN THIS ENVIRONMENT)

Measured results (reproduced in this environment)

Two claims from earlier sections were re-run from the actual project source rather than restated from memory: the lazy-loading memory footprint, and end-to-end mesh timing. Both ran in a single sandboxed CPU container — real numbers from real code, but not a substitute for a multi-machine run on real hardware. Specifics follow.

Lazy-loading memory footprint

A structurally real GPT-2-architecture checkpoint was generated offline — 16 transformer layers, 175.7M parameters, tied embeddings, saved as a genuine `.safetensors` file — and loaded two ways using the project's own loading code.

LOAD PATH	REAL TENSOR BYTES MATERIALIZED	SHARE OF CHECKPOINT
Full load — `AutoModelForCausalLM.from_pretrained`	702,990,080 B (670.4 MiB)	100%
Sliced load — 2 of 16 layers, mid-chain	66,540,032 B (63.5 MiB)	9.5%

Lower than the ~13% figure quoted earlier in this document, because this checkpoint's vocabulary embedding (50,257 × 832) makes up a larger share of total size than in that original test — the embedding table doesn't shrink as layer count grows, so a mid-chain node's 2-of-16 slice of pure transformer blocks comes out to a smaller fraction here. Both figures support the same underlying claim: a node materializes only the layers it owns, not the full checkpoint.

> **Bug found during this reproduction —** the public load_layer_slice() entry point in models/lazy_loader.py currently raises LazyLoadUnsupported for every real deployment shape except one node owning literally all layers. Its final safety check scans every parameter in the full model skeleton for leftover placeholder tensors, rather than only the tensor names that node actually needed — so a normal first/mid/last-chain node always finds (correctly) unrelated layers still unloaded and incorrectly treats that as a failure. The underlying mechanism is sound — _materialize() itself loads every needed tensor correctly, confirmed for first-node, mid-chain, and last-node cases in this run — only the final validation check needs its scan narrowed to the node's own needed-tensor set. The figures above call that narrower, working path directly to measure the real result while this is unresolved.

End-to-end mesh timing

A real registry process, real node processes, and a real coordinator were started as separate local processes communicating over the project's actual TCP wire protocol, then used to generate real tokens from the repo's demo model.

CONFIGURATION	TOKENS	AVG MS/TOKEN	COMPUTE	NETWORK
2 nodes, KV cache on	60	5.0–5.4	~1.9–2.2 ms	~3.0–3.2 ms
2 nodes, KV cache off (`--no-cache`)	60	7.3	3.6 ms	3.6 ms
4 nodes, 1 layer each	60	9.4	3.1 ms	6.3 ms

Two things fall directly out of these runs. First, disabling the KV cache measurably increases both compute (no cached attention to reuse) and total latency, matching Section 6's claim rather than just illustrating it. Second, network overhead scales with chain length — roughly 3.1 ms for 2 hops, roughly 6.3 ms for 4 hops — consistent with a fairly fixed per-hop serialization-and-round-trip cost rather than one growing with total model size.

> **Scope of this test —** all nodes ran as local processes on one machine's loopback interface, not separate physical hardware. "Network" here is real wire-protocol overhead — pickling, socket writes, TCP round trips — genuinely measured, not simulated. It is not a stand-in for real LAN or Wi-Fi latency between distinct machines, which this environment cannot produce. The demo model used (4 tiny transformer layers, untrained, generating character-level gibberish by design — see Appendix) is the project's own test model, chosen because it needs no download and runs instantly on CPU; it is not one of the catalogue models from Section 16.

20 — APPENDIX

Appendix — file guide

FILE	PURPOSE
`run.py`	Terminal live-status view and interactive launcher for all options

`registry/server.py`	HTTP phonebook — nodes announce here, coordinator queries here
`node/server.py`	Node agent — loads a model slice, serves inference, keeps per-request KV cache, hot-reloads
`coordinator/client.py`	Sends prompts, streams output, drives the prefill-then-cached-decode loop
`download_model.py`	Benchmark → recommend → distributed download → switch model
`dashboard/server.py`	Web dashboard — live pipeline view, model switching over HTTP
`models/lazy_loader.py`	Reads only a node's own layers off disk; resolves which shard files to fetch
`shared/protocol.py`	Length-prefixed TCP wire format for tensors between nodes
`config.py`	Model shape and defaults