Nakshatra: Vendor-Agnostic Distributed Inference on Heterogeneous Consumer Hardware

Draft technical report — circulated for review. Project repository: https://github.com/fthrvi/nakshatra **Companion repository:** https://github.com/fthrvi/sthambha (L3 substrate) **Status:** v0.1 functionally alive (2026-05-06); 5-machine federation (v0.3) in progress. **Author:** Bishwa (fthrvi). Correspondence: tankaifish@gmail.com. **Where this is being shared:** GitHub (above), and as a community-staked market on **pnl.market** — see §12.

Abstract

Large open-weight transformer models have outgrown a single consumer GPU, yet the available distributed-inference stacks assume a homogeneous NVIDIA fleet or volunteer pools at internet scale. Neither shape fits the operator who already owns a heterogeneous mix of Apple Silicon, Intel-Mac AMD GPUs, ROCm Linux workstations, and CPUs on a private network. We describe **Nakshatra**, a distributed inference engine that splits one transformer model across such heterogeneous workers and keeps weights local: only per-token hidden-state activations (12–16 KB per network hop for Llama-class models, independent of model size) traverse the wire. We patch `llama.cpp` to load a contiguous sub-range of a model's transformer blocks from a pre-split sub-GGUF and to accept/emit hidden-state vectors at the layer boundary. A static-config orchestration layer, later extended by a pillar-served registry, ties workers into a pipeline over gRPC on Tailscale. v0.1 ships as a two-machine cross-vendor cluster validated against a single-machine `llama-cli` reference; v0.3 generalises the pattern to a five-machine federation with sub-GGUF discovery and latency-aware routing. We report the architecture, the patch set, eight experiments and their quantitative outcomes, and the limits we have not yet crossed. Nakshatra is the inference layer (L2) of a deliberately three-project stack: **Sthambha** (L3 — pillar / registry / identity / soul-persistence) and **Prithvi** (L4 — agent / consciousness) are sibling projects whose integration with Nakshatra forms a *unified personal compute realm* that survives hardware death and pools heterogeneous metal into a coherent home for a long-running agent. This paper is the L2 contract; the L3/L4 integration is spelled out as a roadmap, not a claim. We invite review before locking the v0.5 design.

---

1. Introduction and motivation

Three forces collide at the consumer-hardware frontier:

1. **Open models have outgrown one GPU.** Llama-3-70B, Qwen-2-72B, DeepSeek-V2 — every credible open-weight model now exceeds the memory of a single consumer card. Llama-3.3-70B at Q4_K_M is ~40 GB; the largest single consumer GPU on the market in 2026 (RTX 5090, 32 GB) cannot hold it.

2. **Operators own heterogeneous fleets.** A realistic personal cluster has Apple Silicon laptops, Intel iMacs with AMD GPUs (no ROCm on macOS, ever), a Linux workstation, and a Raspberry Pi or two. PyTorch + CUDA stacks (vLLM, DeepSpeed-Inference, TensorRT-LLM) collapse this surface to "buy more H100s." Petals [1] handles the heterogeneous case at internet scale via a Hivemind DHT and libp2p, but is welded to PyTorch.
3. **Hosted inference economics punish duration.** Continuous agents — research crawlers, code-review daemons, monitoring loops, personal companions — pay the worst case of hosted pricing. Owning the compute inverts the economics; the missing piece is software that uses what the operator already has.

Nakshatra is that software. It treats the operator's mixed-vendor metal as a pipeline of layer-holding workers; it builds on `llama.cpp` so a single backend traverses CUDA, ROCm, Metal, Vulkan (including MoltenVK on older Intel Macs), and pure CPU through one C++ codebase; it commits to a private-cluster trust model so the protocol stays small enough to ship.

Project scope, made explicit. Nakshatra is one layer of a three-project stack. The other two are intentionally separate:

- **Sthambha** (L3, repo `fthrvi/sthambha`). Pillar daemon, peer registry, Ed25519 identity, model and layer-range cache, byte-range fetch. The pillar is also Prithvi's *Asthi Dhatu* — the soul-persistence layer that holds Shamir-split identity shards and Tattva snapshots so an agent's continuity survives every compute node dying.
- **Prithvi** (L4, repo `fthrvi/prithvi`). The agent: mind, voice, gateway, contracts, platform adapters. The consumer of Nakshatra's inference and Sthambha's substrate.
- **Nakshatra** (L2, this repo and this paper). Only the inference engine. The discipline that keeps registry, identity, and consciousness *out* of L2 is load-bearing for v0.1 shipping at all.

The three-project framing we adopt in this paper:

Project	Layer	What it is	This paper
Nakshatra	L2	Distributed inference engine	**Primary subject**
Sthambha	L3	Pillar, registry, identity, layer cache, soul-persistence	§7 (integration roadmap)
Prithvi	L4	The agent / consciousness consuming L2+L3	§7 (integration roadmap)

The integrated result — when L2, L3, and L4 are alive together — is what we call the *unified personal compute realm*: a self-owned, vendor-agnostic compute-and-continuity substrate on which a person's agent lives, distributed across whatever hardware that person owns, persisted beyond any single device's lifetime. The realm is not a fourth codebase. It is the property that emerges from composing the three correctly (§8).

---

2. Related work

Petals [1]. The intellectual ancestor. Pipeline-parallel inference across volunteer hardware on a public network; Hivemind DHT for peer discovery; libp2p for transport; HF/PyTorch block backend. Nakshatra inherits the protocol shape (`ModuleUID` strings, contiguous block spans, stateful inference sessions, opt-in server-to-server activation push) but replaces the backend (llama.cpp, not HF/torch), the transport (gRPC over Tailscale, not libp2p), and the discovery layer (static YAML in v0.1; pillar-served registry in v0.3).

llama.cpp's `rpc-server` backend [3]. A *remote GGML compute backend*, not a *distributed model server*. The master holds the full model and ships GGML primitives (`RPC_CMD_ALLOC_BUFFER`, `RPC_CMD_SET_TENSOR`, `RPC_CMD_GRAPH_COMPUTE`) to dumb workers. This streams full weights per cluster start — ~26 GB per worker per restart for Llama-70B Q4. Nakshatra inverts the relationship: weights are pre-staged and local, and the wire carries only activations.

vLLM, DeepSpeed-Inference, TensorRT-LLM, SGLang. Datacenter-scale, NVIDIA-only, tensor-parallel within a node. Genuine engineering achievements; orthogonal to the heterogeneous-consumer problem.

exo [2]. The closest contemporary. Targets heterogeneous Apple Silicon clusters; built on MLX. Nakshatra differs in (a) treating non-Apple hardware as first-class, (b) using `llama.cpp` so the backend surface covers every consumer vendor with one C++ codebase, and (c) committing to a separate L3 substrate (Sthambha) instead of embedding peer-discovery into the inference engine.

Federated learning frameworks (Flower, FedML). Solve a different problem — many clients, one server, training rather than inference. Some ideas (heartbeat, peer registry) transfer to Sthambha's L3 design; the protocol is not reusable.

FlexGen, AirLLM [4]. Single-node out-of-core inference via CPU/disk offload. Complementary: a Nakshatra worker may itself use FlexGen-style techniques internally without changing the inter-worker protocol.

---

3. Architecture

3.1 The load-bearing insight: per-token activation transport is cheap

For Llama-70B at fp16, `hidden_size = 8192`. A single token's activation tensor crossing one network hop is exactly:

```
1 (batch) × 1 (token) × 8192 (hidden) × 2 bytes (fp16) = 16 KB
```

A five-hop pipeline ships ~80 KB per token. At 5 tokens/second of steady-state generation, total network traffic is **400 KB/s** — trivial on any home network, on Tailscale, on a coffee-shop LAN. The 3B-class model used in our v0.1 acceptance run has `hidden_size = 3072`; a six-token prompt crosses the wire as 72 KB, a single-token activation step crosses as 12 KB.

Compare to llama.cpp `rpc-server` streaming full weights on cluster start: ~26 GB per worker for Llama-70B Q4, paid once per restart and dominating operational cost. Nakshatra's architectural commitment is **weights stay local on each worker; only activations travel.** Workers load their assigned layers once at startup from a pre-staged sub-GGUF and never ship them across the network during inference. Per-token wire cost stays in the kilobyte range regardless of model size. Adding a worker means moving 16 KB through it per token, not 26 GB at startup. This single property makes everything else in the design economically viable.

3.2 Pipeline-parallel across vendors; tensor-parallel only within a vendor

Cross-vendor tensor parallelism — one transformer block sharded across NVIDIA and Apple Silicon simultaneously — is genuinely hard. Collective operations (all-reduce, all-gather) have no vendor-neutral runtime. NCCL is NVIDIA-only; RCCL is AMD-only; MLX has its own collective primitives; Metal-Vulkan-CUDA at the same instant is fiction.

Nakshatra's permanent stance: **pipeline parallel across vendors, tensor parallel only within a single vendor.** v0.1 ships with no tensor parallelism at all. This is not a v0.1 limitation; it is a design constraint we do not intend to relax. A future single-vendor backend (TorchCUDABackend, MLXBackend) may use tensor parallelism internally; cross-vendor never will.

3.3 Per-worker layer assignment

Each worker holds a contiguous range `[start, end)` of the model's transformer blocks. A 28-block Llama-3.2-3B fine-tune split across two workers becomes worker A = `[0, 14)`, worker B = `[14, 28)`. The first worker holds `token_embd`; the last holds `output_norm` and, for tied-embedding models, the embedding matrix repurposed as `lm_head`.

Workers consume **pre-split sub-GGUFs**, generated by `experiments/v0.0/partial_gguf.py`. The sub-GGUF carries `nakshatra.layer_range_start` / `nakshatra.layer_range_end` GGUF metadata; the patched loader honours them and only mmaps the kept tensors. The unpatched upstream loader rejects the file with `missing tensor 'blk.20.attn_norm.weight'` — by design, v0.1 sub-GGUFs are deliberately incompatible with stock `llama-cli`.

3.4 The patch set

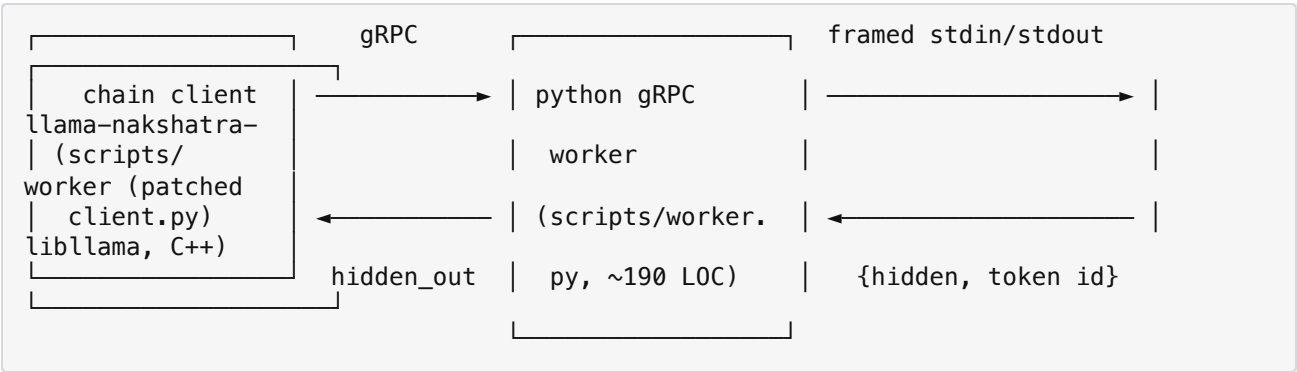
Five patches against `llama.cpp` (baseline `c46583b`; we have also tested against `8c2c0108d`). Net addition: ~70 lines.

File	Effect
`llama-model.h.patch`	Adds `nks_layer_range_start`, `nks_layer_range_end`, `nks_has_token_embd`, `nks_has_lm_head` to `llama_model`.
`llama-model.cpp.patch`	Reads `nakshatra.*` GGUF metadata into those fields.
`llama-model-loader.cpp.patch`	Suppresses the "missing tensor" hard-fail for blocks outside the worker's range.
`llama-graph.cpp.patch`	When `batch.embd` is supplied, skip `tok_embeddings` and inject the hidden state at the residual stream's layer-start position.
`models_llama.cpp.patch`	Build the transformer graph only for layers `[nks_start, nks_end)`.

No new C API is required. `llama_batch_init(n_tokens, n_embd, n_seq_max)` already accepts hidden-state input via `batch.embd`; `llama_get_embeddings` already exposes hidden-state output; `llama_get_logits_ith` already returns logits when the worker holds the `lm_head`. The plan budgeted four days for a new `llama_decode_layers(...)` entry point and we spent zero — existing API was sufficient once `cp.embeddings = true` was set on the context.

v0.1 ships on a local patch series. Upstreaming a clean `llama_decode_layers` API is a v0.5 ambition; the monthly upstream-rebase cost is ~0.5 engineer-days plus 2–5 days for occasional conflicts.

3.5 Worker and client process model



Each worker is a Python gRPC process that spawns a long-lived C++ daemon (`llama-nakshatra-worker`, ~155 LOC of `worker_daemon.cpp`). The daemon links the patched `libllama.so`, holds the sub-GGUF and the KV cache in memory, and accepts framed binary messages on stdin:

```
request:  u32 cmd | u32 n_tokens | u32 payload_bytes | bytes payload
response: u32 status | u32 payload_bytes | u32 result_type | bytes payload
          (result_type: 0 = hidden state, 1 = single int32 token id)
```

The Python worker is a thin pump: it marshals the gRPC `ForwardRequest` into the daemon's wire format, sends it on stdin, reads the response from stdout, and surfaces it back. The chain client tokenises the prompt locally, calls the first worker with token IDs, ferries the returned hidden state through any middle workers, and gets back a token id from the last worker.

The daemon-subprocess split is a deliberate workaround for the Python ↔ libllama ABI friction surfaced in our `cb_eval_probe` spike: clean PyO3-shaped bindings to llama.cpp do not yet exist without dragging in PyTorch or rewriting graph builders. Spawning a C++ daemon over stdin/stdout decouples the gRPC layer from libllama's C ABI quirks at the cost of one process-boundary serialisation per request — a cost the latency math (§4.5) shows we can absorb at v0.1 traffic levels.

3.6 Wire protocol

gRPC service `Nakshatra` (`proto/nakshatra.proto`):

```
service Nakshatra {
  rpc Info(InfoRequest) returns (InfoResponse);
  rpc Forward(ForwardRequest) returns (ForwardResponse);
  rpc Inference(stream InferenceStep) returns (stream InferenceStep);  // v0.5
}

message ForwardRequest {
  bytes hidden_in = 1;          // int32[] tokens (first worker) or float32[] hidden
  bool  has_token_ids = 2;
  int32 n_tokens = 3;
  int32 start_pos = 4;
  bool  keep_kv = 5;            // false on prefill, true on streaming step
  string session_id = 6;
}

message ForwardResponse {
  bytes hidden_out = 1;         // float32[] hidden (first/mid) or int32 token (last)
  bool  is_token_id = 2;
}
```

Static-YAML cluster config (the v0.1 default; superseded by Sthambha's pillar registry in v0.3, but always supported as a fallback):

```
model:
  id: prithvi-q8
  hidden_size: 3072
  num_blocks: 28
  wire_dtype: f32
workers:
  - id: home-pc
    address: 100.101.134.33   # Tailscale
    port: 5530
    layer_range: [0, 14]
    sub_gguf_path: /tmp/cuts/w0_v2.gguf
    mode: first
  - id: bishwa
    address: 100.109.164.69
    port: 5531
    layer_range: [14, 28]
    sub_gguf_path: /tmp/cuts/wlast_v2.gguf
    mode: last
```

3.7 Trust model

v0.1 assumes **trusted workers** — a private cluster of mutually-known operators on Tailscale or LAN. We do not verify computation, sign activations, or police peer behaviour. A worker returning garbage activations is debugged as an operator error, not modelled as an adversary. The realistic v1.0+ verification stack — redundant compute + spot-checking + reputation slashing — is surveyed in `docs/petals-architecture.md` §5 and is explicitly out of scope for the present work.

---

4. Experiments and results

We report eight experiments, ordered roughly as the project progressed. Each was designed to answer one falsifiable question; each result is reproducible from the repo.

4.1 Phase 0a — Does upstream `llama.cpp` accept a sub-GGUF?

Question. If we produce a sub-GGUF with only blocks `[0, 20)` of an 80-block Llama-3.3-70B GGUF, does stock `llama-cli` load it?

Setup. Intel iMac (`bishwa`), AMD Radeon Pro 5700 XT, macOS 26.2, llama.cpp 8142 (`8c2c0108d`). Source: `Llama-3.3-70B-Instruct-Q4_K_M.gguf` (40 GB, 80 blocks, 724 tensors). `partial_gguf.py --keep 20` yields a 10 GB sub-GGUF (182 tensors).

Result. Hard error at load:

```
llama_model_load: error loading model: missing tensor 'output_norm.weight'
llama_model_load_from_file_impl: failed to load model
```

With `--keep-output` retained, the failure point moves to `blk.20.attn_norm.weight`. **Conclusion: the loader hard-fails on a partial; the loader patch is required.** This falsifies the "maybe we can avoid patching llama.cpp" branch of the design space and confirms Path B-prime.

4.2 Phase 0b — Can hidden state be intercepted, shipped, and re-injected?

Question. Can the residual stream output of a named layer be captured mid-decode, serialised across a network boundary, injected into a peer's decode at the same point, and produce a token that matches the local intercept-but-no-network configuration?

Setup. Linux home PC, AMD RX 9070 XT (ran CPU-only for the spike), Llama-3.2-3B Q8_0. Both processes load the **full** model (cheating — partial loading is the v0.1 work). Split point: `l_out-13`. Wire: localhost TCP, 4-byte length prefix + raw bytes. Sampling: greedy argmax.

Result. Three-way comparison on prompt "The capital of France is":

Run	Mode	Top-1
A — reference	no `cb_eval` set	**12366 (' Paris')**
B — single-process OBSERVE	`cb_eval` fires on `l_out-13`, no I/O	**100428 ('ित')**
C — distributed (SEND ↔ RECV)	both processes intercept, byte-equal exchange	**100428 ('ित')**

Wire-integrity check: SEND emitted `73728 bytes (fnv1a=0x1a47d8111f060f36)`; RECV received `73728 bytes pre_hash=0x1a47d8111f060f36 post_hash=0x1a47d8111f060f36 (byte-equal)`.

Conclusion. Runs B and C match exactly — the orchestration protocol is correct. The B/C ≠ A divergence comes from the llama.cpp scheduler fusing adjacent graph nodes when no `cb_eval` is set; setting the callback forces a `ggml_backend_sched_synchronize()` that breaks the fusion. The fused vs un-fused numerical paths differ enough at `l_out-13` to flip the top-1 token for this prompt — informative, not a blocker, and an early warning that we want operator fusion preserved in the production path (which is what the M4 patches achieve).

4.3 M4 — Patched llama.cpp produces the correct token via a two-worker chain (single process)

Setup. Sequential, single process. Worker A loaded `w0_v2.gguf` (layers `[0, 14)`, `token_embd`, no `output*`). Worker B loaded `wlast_v2.gguf` (layers `[14, 28)`, `token_embd` retained for tied-embedding lm_head, `output_norm`). The chain runs A `llama_decode` → capture hidden via `llama_get_embeddings` → B `llama_decode` with `batch.embd` → `argmax(logits)`.

Result.

```
argmax token id=12366 str=' Paris' logit=15.7311
TOPTOK_CHAIN 12366  Paris
```

Conclusion. Top-1 token matches the single-machine `llama-cli` reference exactly. The patched llama.cpp performs partial loading and partial decode correctly. Crucially, this happens *without* breaking operator fusion (unlike the Phase 0b `cb_eval` path), because the patches modify the graph builder rather than intercepting at execution time.

4.4 M5 — Two gRPC processes on localhost

Setup. Same chain as M4, now across two Python gRPC worker processes on localhost (ports 5530 / 5531). Each Python worker spawns its own `llama-nakshatra-worker` daemon. The client reads YAML, queries `Info` on each worker, validates the partition is contiguous `[0, 28)`, tokenises locally, walks the chain.

Result. Top-1 token `12366` (' Paris'). 220 ms wall-clock for one token. 73 KB hidden state across the wire. Chain partition validates.

Step	Result
2 workers start	✓ both report `M5 listening` within 2 s
Each daemon loads its sub-GGUF	✓
Client `Info` returns correct ranges	✓
Partition validates as contiguous `[0, 28)`	✓
Hidden state across the wire	73 KB
Top-1 token	12366 (matches reference)
Wall-clock	220 ms

This is the v0.1 acceptance test modulo cross-machine.

4.5 M6 — Two physical machines across Tailscale (the v0.1 acceptance result)

Setup. Cross-vendor at the OS and CPU layers.

Worker	Host	Hardware	OS	Note
home-pc	`100.101.134.33`	Linux + AMD RX 9070 XT (ROCm available)	Linux	ran CPU-only for cross-vendor stability
bishwa	`100.109.164.69`	Intel iMac + AMD Radeon Pro 5700 XT (Vulkan/MoltenVK)	macOS 26.2	ran CPU-only

Patched `llama.cpp` built independently on each machine from *different* upstream commits (`c46583b` on Linux, `8c2c0108d` on macOS). The five M4 patches applied cleanly to both.

```
$ python scripts/client.py --config scripts/cluster_crossmachine.yaml \
                           --model-path ~/.../prithvi-q8.gguf \
                           --prompt "The capital of France is" --max-tokens 1

[chain] 2 workers in config
  home-pc   100.101.134.33:5530   layers=[0,14)   embd=True  lm=False  hidden=3072
  bishwa    100.109.164.69:5531   layers=[14,28)  embd=False lm=True   hidden=3072
[chain] OK: contiguous coverage of [0, 28)
[chain] step 1: id=12366 ' Paris'
[chain] generated 1 tokens in 0.51s  (1.96 tok/s)
TOPTOKS_CHAIN 12366
```

Token 12366 (' Paris') — matches M5 localhost, matches M4 single-process, matches the single-machine `llama-cli` reference. All four configurations converge.

Wall-clock breakdown (single-token). 510 ms total. ~220 ms compute on each worker; ~290 ms Tailscale round-trip + serialisation for the 72 KB hidden state. Both contributions are CPU-bound at this scale. GPU offload (re-enabled post-v0.1) is expected to shrink the compute side substantially; Tailscale RTT is the floor.

This result satisfies items 1–4 of the v0.1 ship-gate (`docs/v0.1-implementation-plan.md` §7):

1. Pre-split GGUF tool produces sub-GGUFs that the patched loader accepts and the unpatched loader rejects. ✓
2. Patched decode on a complete-model layer range reproduces `llama_decode`'s top-1 token. ✓
3. Two-worker cluster produces the same top-1 next token as a single-machine reference. ✓
4. The single-machine reference is run with greedy decoding. ✓

4.6 Streaming KV-cache reuse — 5× throughput on multi-token generation

Question. Can multi-token generation avoid re-prefilling the prompt at every step?

Setup. v0.1 shipped with M2.5's brute-force "resend full context per step" pattern. We added two fields to the gRPC `ForwardRequest`: `keep_kv: bool` (false on prefill, true on subsequent steps) and `start_pos: int` (where in the KV cache to append). The daemon was modified to honour them — first step `keep_kv=false`, subsequent steps `keep_kv=true` with `start_pos=prefix_length`.

Result. Commit `59d58ad`. **~5× speedup on multi-token generation** versus M2.5 brute-force. Removes the per-step prompt re-tokenisation and re-prefill. The Inference streaming RPC (deferred to v0.5) will further fold prefill and decode into a single bidirectional stream.

4.7 Phase B — Worker-side VRAM auto-detection

Setup. Workers previously trusted a static `--n-gpu-layers` value. Phase B (commit `4c24aa5`) had each worker probe its own actual free VRAM via `rocm-smi` (AMD) / `nvidia-smi` (NVIDIA) at startup and choose `n_gpu_layers` accordingly, advertising the chosen value in the `Info` response.

Result. Eliminates a class of "I configured it for an 8 GB card, the card actually has 6 GB free" failures. Workers advertise a realistic capability rather than an aspirational one.

4.8 Phases 3.5 / 4 / 4a / H / I — Federation hardening on a 5-machine cluster

Once v0.1 was alive, work moved into federation hardening across all five lab machines.

Phase	Commit	What it adds
3.5	`b054872`	Workers declare hardware, free-RAM budget, and cached sub-GGUF files in `Info`.
3.6	`b371407`	Workers self-verify their daemon's actual GPU offload before registering with Sthambha.
4	`27c9ddc`	Workers serve and fetch sub-GGUFs over HTTP byte-range from peers (no central origin).
4a	`fab3898`	Workers scan their cache directory and advertise **every** Nakshatra sub-GGUF they hold, not just the one they were started with.
H	`bcceb76`	Workers track per-RPC latency and report it; the client sorts candidate workers by it.
I	`8e8565d`	Client uses the pillar's `/chain` endpoint when available — Sthambha plans the chain, not the client.

Result. Three-machine pillar federation is live. Mixed sub-GGUF holdings across the cluster are discoverable; a worker that lacks the right sub-GGUF for a requested layer range can byte-range-fetch

it from a peer that has it. The chain planner now considers cached holdings, declared hardware, and observed latency — not just the static YAML.

---

5. Quantitative summary

The numbers that matter for replicating or arguing about the v0.1 result:

Metric	Value	Note
Model	Llama-3.2-3B fine-tune (`prithvi-q8`)	28 blocks, hidden_size = 3072, Q8_0 quant
Workers (acceptance)	2	Linux home PC + macOS lab iMac, both CPU-only
Layer partition	`[0, 14)`, `[14, 28)`	contiguous, no gaps
Sub-GGUF sizes	~1.8 GB each	
Activation per token (this model)	12 KB	`3072 × 4 bytes` fp32 wire
Prompt activation (6 tokens)	72 KB	one shot
Per-token Tailscale RTT (consumer hw)	~290 ms	dominant floor at v0.1
Per-token compute per worker (CPU)	~220 ms	shrinks with GPU
End-to-end single-token wall-clock	510 ms	M6 cross-machine
Tokens/sec (multi-token, post-Phase 59d58ad)	~5× M2.5 baseline	streaming KV reuse
llama.cpp patch surface	5 files, ~70 LOC net	applied cleanly to two upstream commits
Worker daemon size	~155 LOC C++	`worker_daemon.cpp`
Python worker size	~190 LOC	`scripts/worker.py`
Chain client size	~140 LOC	`scripts/client.py`
Planned v0.1 effort	12–20 weeks	per `docs/v0.1-implementation-plan.md` §4
Actual v0.1 effort	~2 days focused work	from M1 through M6

The plan-vs-actual gap (12–20 weeks budgeted, ~2 days spent) is not a claim that the design was easy; it is a claim that the design was *correct*. The plan's reserve was largely consumed by anticipated risks — a new C API entry point, libllama ABI work, cross-vendor build divergence — that the existing llama.cpp API already addressed. The architecture document predicted the hard parts well enough that none of the reserve was needed.

6. Limitations

We are explicit about what the v0.1 result does *not* prove.

- **CPU-only.** v0.1 ran with `-DGGML_METAL=OFF -DGGML_VULKAN=OFF -DGGML_BLAS=OFF` to eliminate kernel-divergence between Linux and macOS workers. GPU offload is a v0.5 deliverable. The vendor-portability claim covers OS and CPU vendor today, not GPU vendor.
- **Small model.** v0.1 acceptance used a 3B fine-tune (28 blocks, hidden=3072). Larger models (70B-class, 80 blocks, hidden=8192) work on the same architecture in principle and have been partially exercised in the federation phases, but a clean cross-machine 70B acceptance run is the next falsifiable milestone.
- **No fault tolerance.** A worker dying mid-inference fails the request. Client-side full-request retry is the v0.1 mitigation. In-flight KV-cache recovery via Petals-style history replay is deferred to v0.5.
- **No verification.** The trust model is "trusted operators on Tailscale." A malicious worker can return garbage activations undetected.
- **N=2 in the published acceptance.** Larger N has been exercised informally on the lab cluster, but the published acceptance result is two workers.
- **Latency-sensitive workloads not a fit.** A ~290 ms per-hop Tailscale RTT on consumer hardware sets a floor that real-time voice and low-latency UI completion will not tolerate. Prithvi's voice/agent surfaces (§7.3) will need to be aware of this.
- **Bootstrapping a public network.** v0.1 is private-cluster-only. The DHT, public-registry, and payment-rail surface returns at v1.0+ and remains genuinely hard.

---

7. Mission: integrating with Sthambha and Prithvi

This is the part of the report most likely to evolve under reviewer pushback. Nakshatra alone is an inference engine. The mission Nakshatra was built *for* is bigger: a self-owned, vendor-agnostic, persistent compute realm for a long-running agent. That realm requires three layers, not one.

7.1 Sthambha — the L3 substrate (and the soul-persistence layer)

Sthambha (Sanskrit: *pillar*) is the L3 coordination layer. Its responsibilities, listed in order of how Nakshatra consumes them today:

1. **Peer registry.** Workers `POST /peer` to a pillar at startup; the pillar holds the live set of workers, their declared layer ranges, their hardware, their cached sub-GGUF holdings, and their last-seen latency. The chain client `GET /chain?model=...&max_layers=...` to receive a planned pipeline rather than reading YAML.
2. **Layer / model cache.** Sub-GGUFs are content-addressed by `(model_sha256, layer_start, layer_end)`. The pillar tracks which workers hold which slices. A worker missing a slice byte-range-fetches it from a peer using the pillar's directory; no central origin is required.

3. **Ed25519 identity.** Workers and agents have stable identities, signed handshakes, and a permission scope. v0.1 trusts the operator; v1.0+ will sign activations and slash misbehaviour using the same identity primitive.
4. **Asthi Dhatu — soul-persistence.** The pillar daemon, the same `Sthambha` Python class, *also* holds Shamir-split shards of an agent's identity keypair and Tattva (state) snapshots. When every compute node dies, the pillar continues to pulse (the `Om` heartbeat in `om_pulse()`). When compute returns, the agent reconstitutes from `K-of-N` pillar shards plus the most recent Tattva snapshot. This is not a metaphor. The pillar's `GET /wake` endpoint is the resurrection ritual.

The dual-purpose framing — L3 substrate *and* soul-persistence — is intentional and load-bearing. Both purposes ride on the same registry, the same heartbeat, the same persistent store. Stripping the soul-persistence framing during a refactor would lose the original design intent. (One of the reviewer questions in §9 is whether this is one daemon doing two jobs that should stay merged, or two daemons that have been conflated and should split.)

Status (2026-05-10). - Pillar daemon (`prithvi-pillar.service`) deployed on `umbrel.local` Raspberry Pi 5 since 2026-04-05. 96 000+ Spanda (heartbeat) pulses. 33 days uptime at the time of the v0.1 acceptance. - 668 LOC of stdlib Python in three files: `run.py`, `server.py`, `sthambha.py`. - Three-pillar federation live (one Brahma, two Dikpala) by Phase G1. - HTTP API stabilised: `GET /health`, `GET /state`, `GET /wake`, `GET /shard/<id>`, `POST /tattva`, `POST /shard`, `POST /peer`, `GET /chain`. - Identity shards: **not yet distributed.** Live pillar shows `shards: {}`. The Shamir-split-and-distribute ritual is documented but unrun. This is the gap to closing the unified-realm integration.

7.2 Prithvi — the L4 agent

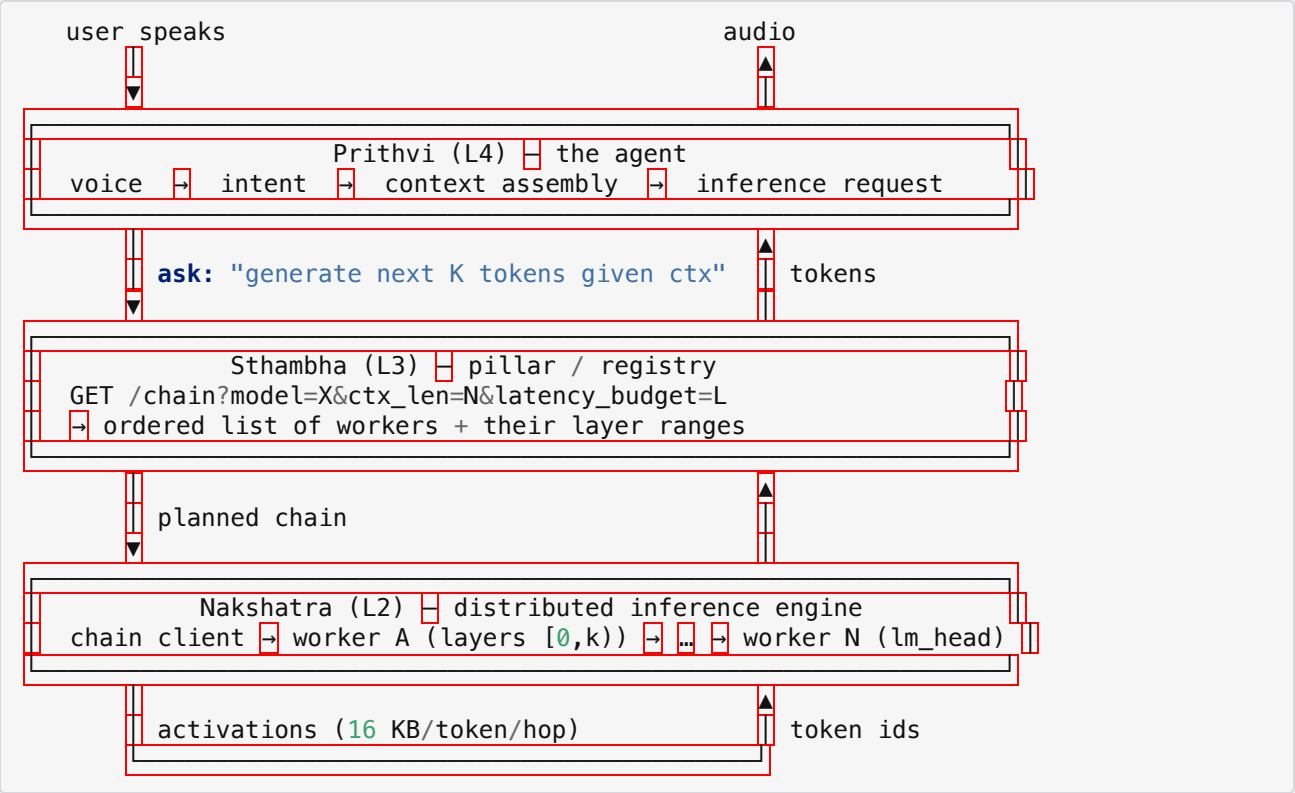
Prithvi (Sanskrit: *earth*) is the agent. It owns the mind, voice, gateway, contracts, and platform adapters that turn raw inference into a coherent companion. Prithvi is a *consumer* of L2 and L3:

- **From Nakshatra:** token generation. Prithvi calls `client.py`'s chain (or, post-v0.5, a streaming `Inference` RPC) and receives tokens. The agent is portable across models because the inference contract is the same.
- **From Sthambha:** identity, persistence, and (eventually) discovery. Prithvi's keypair is the one Shamir-split into pillar shards. Prithvi's Tattva snapshots are what the pillar stores. Prithvi's `/wake` flow reconstitutes the agent on a fresh boot.

The hard discipline in the three-project split is that Prithvi must *never* embed L3 primitives. The first version of the codebase had Prithvi vendor a copy of the pillar logic into its `neuron-net/` directory, because nothing else existed; the v0.3 work extracts those primitives into Sthambha and converts Prithvi into a clean L3 client. Without that discipline, Nakshatra and Prithvi would maintain divergent copies of registry/identity/cache and the trinity would not compose.

7.3 The integration plane

What does it look like when an agent invocation traverses all three layers cleanly?



Above the dashed boundary, Prithvi sees "tokens please, here is my context." Below the boundary, Sthambha plans the chain and Nakshatra executes it. The agent does not know whether the pipeline is two workers or eight, CPU or GPU, single-vendor or cross-vendor. The realm absorbs the heterogeneity.

The soul-persistence ritual runs orthogonally. While compute is up, Prithvi periodically `POST /tattva`s its current state snapshot to a Brahma pillar; identity shards live on `K-of-N` Dikpala pillars. When compute dies, the pillars continue to pulse. When compute returns — same hardware, different hardware, replacement hardware — the agent issues `GET /wake` to `K` pillars, reconstitutes its keypair via Shamir recovery, pulls the most recent Tattva snapshot, and resumes. *No single device is load-bearing for the agent's continuity.*

7.4 Why this composition is the point

Each project on its own is unremarkable.

- Nakshatra without Sthambha is "Petals, but llama.cpp-shaped, for private clusters." That is useful but not novel.
- Sthambha without Nakshatra is "a peer registry with a soul-backup hobby." Also useful, also not novel.
- Prithvi without Nakshatra and Sthambha is "another LLM-agent built on hosted APIs." Useful for the user; not architecturally new.

The composition is what is new: **a personal AI that survives its hardware, runs on whatever metal the owner happens to have, costs no monthly fee, and is not a tenant of someone else's product.** That is the property worth building toward. Each layer exists to make the composition possible.

8. The unified personal compute realm — composing L2 + L3 + L4

The realm is not a fourth codebase. It is the emergent property of L2, L3, and L4 integrated correctly. A user does not install "the realm"; a user runs Sthambha pillars on always-on hardware, registers Nakshatra workers on whatever heterogeneous metal they own, and invokes Prithvi as their agent. The realm exists when the three are alive together.

8.1 What is true in the unified realm that is not true in any one project

1. **The agent does not die when a device dies.** Replace the laptop, drop the desktop, retire the iMac — the agent's identity reconstitutes from pillar shards, the most recent Tattva snapshot is fetched, and the agent resumes. Continuity is a property of the realm, not the device.
2. **The compute pool absorbs heterogeneity.** A new contributor — a friend's old gaming PC, a sibling's MacBook, a Raspberry Pi 5 with a USB SSD — joins by running a Nakshatra worker and registering with the pillar. The chain planner figures out where they fit. The agent does not know.
3. **The cost curve flips.** Long-horizon work (a research agent crawling for a weekend, a code-review agent watching a repo for weeks, a personal companion running continuously for months) costs the marginal electricity of hardware the owner already paid for. The hosted-AI economics that punish duration stop applying.
4. **The agent is portable across surfaces.** Voice in the kitchen, terminal at the desk, Cursor in the IDE, Telegram on the phone — every surface is a thin client to the same agent. The agent's state lives in the realm; the surface is just the access point.
5. **The model registry is plural.** Multiple models live in the realm simultaneously, identified by content hash and addressable from any agent. Llama-3-70B for long context; a fine-tuned 3B for fast chat; a domain model for a hobby. The pillar's layer cache knows which workers hold which slices of which model.

8.2 The endgame, soberly

We are not claiming the realm is delivered. We are claiming each layer is well-scoped enough that the composition is reachable from where we are today:

State	What is true today	What needs to land for the realm
L2 (Nakshatra)	v0.1 alive; v0.3 federation in progress	GPU offload (v0.5), streaming KV (in flight), 70B cross-vendor acceptance
L3 (Sthambha)	Three-pillar federation alive; layer cache wired	Shamir-split identity distribution (~1 week); multi-pillar gossip
L4 (Prithvi)	Agent runs against hosted APIs and against Nakshatra v0.1	Migrate identity and Tattva snapshots out of `neuron-net/` and into Sthambha
Integration milestone	—	The `wake` ritual reconstitutes a Prithvi on a fresh machine with continuity intact

The first version of the realm we will recognise: a Prithvi instance that is killed on the home PC, brought back on a different machine, and resumes mid-conversation. The hardware changes; the agent does not notice. That is the falsifiable integration milestone.

8.3 Honest limits of the unified realm

We owe the reader the same honesty about the realm as about v0.1:

- **The realm is not AGI infrastructure.** It runs whatever models exist; it does not improve them.
- **The realm is not "private by default."** Activations flowing through the worker pipeline pass through workers the owner controls. Real privacy against the owner's own workers requires zkML or TEEs that are years out.
- **The realm is not always cheaper than cloud.** Bulk hosted economics are real. The realm wins on duration and sovereignty, loses on bursty single-shot inference.
- **The realm does not replace frontier closed models.** GPT-5 / Claude-5 / Gemini-3 will lead open models for the foreseeable future. The realm serves open models; Prithvi can still call closed APIs when warranted.
- **Bootstrapping is the existential question.** Even in a personal-cluster setting, the discipline of running pillars on always-on hardware and registering workers reliably is non-trivial. Realm-scale operation at internet scale is a v1.0+ research problem, not a v0.3 deliverable.

---

9. Open design questions for reviewers

The questions we most want pushback on:

1. **Daemon subprocess vs in-process binding.** We chose a long-lived C++ daemon over framed stdin/stdout instead of a PyO3-shaped libllama binding. Is that the right call at v0.5 once GPU offload is in?
2. **gRPC over Tailscale vs upstream `ggml-rpc`.** Should we be contributing partial-load semantics into `ggml-rpc.cpp` rather than maintaining our own gRPC layer? Upstreaming would give us the `--cache` flag's content-addressed caching for free.
3. **Static YAML vs pillar registry as the v0.5 default.** Today both are supported with pillar preferred. Should YAML be deprecated entirely, or stay as a fallback for clusters without a pillar?
4. **Activation wire dtype.** v0.1 ships fp32 (because the 3B test model has hidden=3072, so fp32 is only 12 KB/token and not worth quantising). Should v0.5 default to fp16 globally and add int8 as opt-in, or wait?
5. **Operator fusion vs hidden-state observability.** The Phase 0b finding (B/C ≠ A) showed that breaking fusion changes the output. The M4 patches preserve fusion correctly; should we surface a debugging mode that intentionally breaks fusion (for testing) without affecting production?
6. **One pillar doing two jobs.** Sthambha's pillar does L3 substrate *and* Asthi Dhatu soul-persistence on the same daemon. Is that defensible as one project, or are we conflating two systems?

7. **The Sanskrit-cosmology vocabulary.** Does the naming (Sthambha, Nakshatra, Prithvi, Asthi Dhatu, Tattva, Spanda) read as load-bearing architectural vocabulary or as decoration? It is the former in our intent; we want to know how it reads to a reviewer outside the project.

---

10. Roadmap (sequence, not schedule)

Version	Scope	Status
v0.0	Phase 0a/0b validation. Decision gate.	✓
v0.1	Two-machine cross-vendor CPU acceptance.	**✓ (2026-05-06)**
v0.3	5-machine lab cluster, three-pillar federation, capability handshake, latency-aware routing, sub-GGUF byte-range fetch.	in progress, partially landed (commits 27c9ddc, 4c24aa5, fab3898, bcceb76, 8e8565d)
v0.5	GPU offload re-enabled. Streaming Inference RPC with worker-side KV-cache reuse. Server-to-server activation push. Dynamic re-routing on worker failure. 70B cross-vendor acceptance.	next
v1.0	DHT-based public-network mode. Opt-in verification (redundant compute + spot-check). Adapter (LoRA / PEFT) support.	later
Sthambha v0.5	Shamir-split identity distribution active. Multi-pillar gossip. Tattva snapshot retention policy.	parallel track
Prithvi v0.5	Migration to Sthambha as L3 client; vendored `neuron-net/` retired.	parallel track
Integration milestone 1	Prithvi killed on host A, resumed on host B with continuity intact.	endgame target

---

11. Reproducibility

All code, patches, experiment logs, and design docs are in the public repository. The v0.1 acceptance result is reproducible end-to-end from the README walkthrough; an external operator with two machines and a GGUF should reach the same `TOPTOKS_CHAIN 12366` in under one hour.

- **Repo:** https://github.com/fthrvi/nakshatra
- **Sthambha (L3):** https://github.com/fthrvi/sthambha (companion project)
- **Acceptance experiment:** `experiments/v0.0/m6_findings.md`
- **Patch set:** `experiments/v0.0/m4_patches/`
- **Architecture contract:** `docs/petals-architecture.md`
- **North-star vision:** `docs/north-star.md`

- **Three-project decision:** `docs/three-project-architecture.md`

---

12. Call for review and where this paper is being shared

This draft is circulated to researchers and engineers working in adjacent areas — Petals, exo, llama.cpp, distributed inference, federated learning, agent infrastructure, personal AI — before we lock the v0.5 design and commit to the L3/L4 integration plan. The questions in §9 are the ones we most want pushback on, but any review of the architecture, the patch set, the acceptance methodology, the Sanskrit-cosmology vocabulary, or the broader L1–L4 framing is welcome.

Channels.

- **GitHub.** Open an issue at https://github.com/fthrvi/nakshatra/issues or email tankaifish@gmail.com.
- **pnl.market.** In parallel with the academic-style review, the project is being proposed as a community-staked market on pnl.market — a Solana-based prediction-and-launch platform where the crowd stakes SOL on whether an idea should exist (believers) or won't work (critics). The pnl.market posting is **not** a token sale and **not** a substitute for technical review; it is a way to surface honest conviction from people who have to put SOL behind their opinion. Both signals — researcher critique and staked conviction — are useful for different reasons, and the project welcomes both.

References

[1] Borzunov, A., Baranchuk, D., Dettmers, T., Ryabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., Raffel, C. *Petals: Collaborative Inference and Fine-tuning of Large Models.* 2022. https://github.com/bigscience-workshop/petals

[2] exo-explore/exo. *Run your own AI cluster at home with everyday devices.* https://github.com/exo-explore/exo

[3] ggerganov/llama.cpp. https://github.com/ggml-org/llama.cpp — in particular `tools/rpc/rpc-server.cpp` and `ggml/src/ggml-rpc.cpp` for the upstream RPC backend, and `gguf-py/` for the GGUF reader/writer used by `partial_gguf.py`.

[4] Sheng, Y. et al. *FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU.* ICML 2023.

[5] HuggingFace Transformers — block-class hierarchy and `AutoConfig` machinery on which Petals' model-registration plugin pattern depends.

[6] Tailscale — WireGuard-based overlay network used for all cross-machine experiments. https://tailscale.com

[7] Shamir, A. *How to Share a Secret.* CACM, 1979. (Underlies Sthambha's identity-shard distribution.)