Kafila: Serving Large Language Models on a Trusted Set of Heterogeneous Commodity Machines

Murtaza Rangwala, Richard O. Sinnott and Rajkumar Buyya
Quantum Cloud Computing and Distributed Systems (qCLOUDS) Lab
School of Computing and Information Systems
The University of Melbourne, Australia
Email: mrangwala@student.unimelb.edu.au, {rsinnott, rbuyya}@unimelb.edu.au

***Abstract*—Between them, the members of a research group or a circle of friends own several consumer computers, none large enough to run a capable large language model. Existing systems pool such capacity across open swarms anyone may join, which a group admitting only trusted machines cannot use. Bounding membership removes what they depend on: a swarm holds each part of the model on several peers and routes around a slow one. A bounded session must use every device it admits. Its pipeline advances at the pace of whichever device received a share it cannot serve quickly, so the division has to be right before serving begins. We propose Kafila, whose protocol assembles a ring from behind NATs, preferring direct paths and relaying where traversal fails, while its planner measures each device's memory bandwidth, capacity and reachability, divides the model exactly for a fixed ring order, and places the head, which holds the embedding and output projection, together with that division rather than beforehand. On machines with different capabilities across three fleets, from a shared LAN to five devices spanning two continents, Kafila shortens the slowest pipeline stage by up to $5.2\times$ against the even split of pipeline parallelism, as in GPipe, and up to $3.0\times$ against the memory-proportional split of personal-device inference, as in exo, keeps 75 to 87 per cent of the committed hardware doing work where those divisions fall below half, and serves a model no uniform split can place on the fleet at all. What that is worth to a user depends on how much of a token is computation rather than network. Where the members share a network the same division returns $1.56\times$ the throughput of a uniform split and $1.25\times$ of a memory-proportional one, and under four concurrent users that lead compounds to $3.2\times$ rather than fading, each user served at almost the rate of one.**

***Index Terms*—device-to-device coordination, large language model inference, heterogeneous machines**

I. Introduction

Between them, the members of a research group or a circle of friends own several computers capable of running neural networks: laptops, desktops with discrete graphics cards, the odd workstation. None is large enough on its own to run a capable large language model (LLM). This is a familiar class of problem in pervasive computing: the useful resource is the set of devices a group already has, not any one of them, and the task is to coordinate them rather than to acquire more.

Our motivating application is a model that answers questions about a person's own documents, messages, and health records [1], [2], the content people are least willing to send elsewhere. The most capable models are reached only through APIs [3], so using them means sending that content to infrastructure the user cannot audit. Open-weight models close the capability gap [4] but not the hardware gap: a competitive model does not fit in the memory of any single device the group owns [5], though it may fit in all of them together. Turning that aggregate into a usable model server is a device-to-device coordination problem.

One line of work closes the hardware gap by pooling capacity across the Internet, splitting a model's layers across an open swarm of volunteers [6]–[8]. Such a pool assembles many small machines into an effectively larger one, and can grow without bound because anyone may join. The property that makes such a pool large, however, also makes it unsuitable here. A user who wants to run a model only on machines they trust cannot use an open swarm. Restricting membership is not a configuration choice in these systems; it removes the assumption they are built on. Two consequences follow, the second of which is the subject of this paper.

The first concerns who holds a user's activations. In an open swarm they travel to whichever volunteer holds the corresponding block, and a server holding them can reconstruct the input with high fidelity, even against common perturbation defenses [9], [10]. Confining the group to invited participants keeps every block with an invited peer, so activations reach no unknown volunteer, and a relayed edge that must cross the rendezvous crosses it end-to-end encrypted, as ciphertext the rendezvous cannot read; the protocol prefers direct paths all the same (Section IV-B), and Section VII-A measures how often it relays per fleet. We treat the block-level guarantee as a premise of the design rather than a contribution.

The second consequence is that a bounded group is harder to run a model on quickly. An open swarm tolerates a slow participant because it has redundancy: many peers hold each block, so a client can be routed around a poor one, which is what Parallax's scheduler does [8]. A bounded group of five invited devices has no such freedom. Every admitted device must be used however slow, and a pipeline advances at the pace of its slowest stage. Existing systems do not plan the forward pass against measured per-device capability. That slowest stage is therefore whichever device received an equal share of the model rather than the share it could sustain. Throughput suffers even where the aggregate hardware would allow better. For instance, Petals reports roughly one decoding step per second for a 176-billion-parameter model [6]. A bounded group therefore does not need a better route through

a large pool. It needs a division of the model that suits the devices it actually has.

We call such a bounded group a *session*: it is assembled on demand to serve one model and dissolves when its last member leaves. Membership is defined by relationship rather than physical proximity: at one extreme, a session is a single household's devices, necessarily co-located; at the other, it is a research group's workstations, or a group of friends' machines, spread across cities or continents. Scoping by relationship rather than by proximity, however, is what makes a session hard to realize, and two difficulties follow, both of which this paper addresses. First, the devices sit behind independent home or institutional Network Address Translators (NATs), and cannot reach one another until they are told how. A pipeline needs every adjacent pair in the ring to connect. One unreachable pair therefore does not merely slow a session down; it stops the session from forming at all. Second, the devices differ by an order of magnitude in memory and in memory bandwidth, and no redundancy exists to absorb that difference. The division of the model therefore decides both how fast a session runs and whether it runs at all. We present Kafila, a session protocol and planner for exactly this setting. It makes the following contributions:

- **A session protocol** that forms a pipeline entirely from behind NAT, with no router configuration, using an always-on rendezvous service for signaling only. It prefers hole-punched direct paths and relays only where traversal fails.
- **A planner** that decides block assignment and head placement from each device's measured memory bandwidth and capacity, over a ring ordered from measured reachability. Placement is solved exactly for a fixed ring order by chain-on-chain partition, head placement jointly with that division rather than beforehand, and a session for which no assignment fits is declined.
- **An evaluation** across three fleets, from a shared LAN to five devices on two continents, on models up to 32 billion parameters. Dividing against measured capability, the planner shortens the slowest pipeline stage by up to $5.2\times$ where capability-blind divisions leave much of the hardware idle, and serves a model no uniform split can place at all.

The rest of the paper is organized as follows. Section II situates Kafila relative to the open-swarm and pipeline-parallelism literature. Section III defines the session model. Section IV describes the protocol that forms a session from behind NAT. Section V presents the planner. Section VI describes the implementation, the testbed, and the NAT emulation the protocol results rest on. Section VII reports the evaluation. Section VIII concludes and states what remains open.

II. Related Work

A. Open-swarm collaborative inference

Petals [6] is closest in spirit. It partitions a model across volunteer machines and pipelines requests through them, the same workload structure Kafila targets, but differs in population and hence in guarantees. Petals' swarm is open and unbounded, tolerating churn through redundancy. A departing server is routed around because another elsewhere holds the same block. A session, on the other hand, has no redundant capacity to route around with. This is not a design decision that could be reversed. A user cannot both confine a model to machines they trust and retain a pool large enough to hold each block several times over, since the number of machines a person has a relationship with is small. The problem is therefore to make a small, fixed membership functional, not to make an unbounded population resilient. SWARM parallelism [7] and Parallax [8] extend the open-swarm model to training and to explicitly heterogeneous, multi-datacenter GPU pools with dynamic scheduling, but retain the same assumption of an unbounded, changing peer set with no guarantee over ring adjacency. Both mitigate the resulting performance variance by routing and scheduling at runtime rather than by planning in advance. This is the reasonable choice for an open population, which changes too fast for a plan to remain valid. It is also what a bounded group cannot borrow. Routing around a poor peer requires a peer to route to, and a session has none, so the quality of the initial division is the only lever available.

B. Pipeline parallelism over heterogeneous, imperfectly connected machines

Pipeline parallelism was developed for the datacenter, and its partitioning rules are the ones practitioners still inherit. GPipe [11] divides a network into stages of equal layer count, which is sound when the devices are identical. exo [12], which assembles a ring across phones, laptops and desktops, instead gives each device a share proportional to its memory, which is sound when memory and reading speed rank alike. A session guarantees neither. Its devices differ in speed, and the one with the most memory is often not the fastest, so both rules hand blocks to a device that cannot serve them at the rate the division assumes.

PipeDream [15] instead minimizes the slowest stage by dynamic programming, which is Kafila's objective, and PipeEdge [13] and PipePar [14] extend that to unequal devices. All three assume a complete graph and a deployment administered as one system. A session has neither: its pairs can communicate only sometimes, and which pairs is unknown until measured. Two departures follow. Ring order is chosen from measured reachability rather than given. And stages are not interchangeable, since the head holds the embedding and the output projection, so it costs more per token and holds fewer blocks at equal memory, an asymmetry Kafila carries into both the per-device cost function and the per-device block limit so that head placement is solved together with the division.

The decomposition rests on Pinar et al. [16], who show that *chain-on-chain* partitioning, a contiguous chain divided across heterogeneous processors whose order is fixed, is exactly solvable in polynomial time, while choosing that order as part of the same problem is NP-complete. Kafila fixes the ring order

TABLE I
COMPARISON WITH RELATED SYSTEMS

System	Workload	A slow or absent participant	Connectivity graph	Per-stage cost obtained from	Ordering by reachability	Head placed with the division
Petals [6]	LLM inference	Routed around at request time	Complete	Throughput measured at runtime	×	×
SWARM [7] / Parallax [8]	LLM inference	Rescheduled at runtime	Complete, unequal bandwidth	Throughput measured at runtime	×	×
GPipe [11]	Training	Does not arise; deployment is provisioned	Complete, uniform speed	Layer count	×	×
exo [12]	LLM inference	Slows the ring; not addressed	Complete, on one network	Device memory	×	×
PipeEdge [13] / PipePar [14]	Encoder inference; training	Does not arise; deployment is provisioned	Complete, unequal speed	A profile taken with the weights resident	×	×
Kafila (this work)	LLM inference	Cannot be avoided, so it is planned for	Incomplete and unequal	A bandwidth probe, before any weight is fetched	✓	✓

first, from measured reachability, then solves block placement exactly.

C. NAT traversal

Hole punching establishes a path through a NAT by having both peers send outbound packets, which the middlebox then learns to forward responses through. Ford et al. [17] documented and analyzed it for UDP and TCP. Its building blocks are standardized in STUN [18] and ICE [19], and the NAT behavior taxonomy Kafila reasons about follows RFC 4787 [20]. Trautwein et al. [21] measured hole punching across more than 85,000 IPFS networks. Roughly 70% of attempts succeed once relay reservation and address discovery complete. That is an appropriate anchor for an open population of arbitrary peers. For a session's smaller and more idiosyncratic membership it is an assumption to verify rather than to inherit.

D. Summary

Table I sets these systems against the single question that separates them: what happens when a participant is slow or absent. A swarm reschedules around it; a provisioned deployment never meets it. A session has neither a spare peer nor an administrator, so the division of the model is decided once, in advance, from what the devices happen to be and where they happen to sit. That is why Kafila must get the division right before serving begins, with no runtime recovery to fall back on, and why it must order members by measured reachability, a step a cluster never needs.

III. THE KAFILA SESSION MODEL

A *session* is a bounded set of devices, assembled on demand, that together serve one model for as long as its members remain. Three roles participate. The *rendezvous* is one always-on component, reachable outbound by every device, that carries session registration and the signaling required for NAT traversal. It holds no model weights and performs no inference, so it needs neither an accelerator nor substantial compute, and a small cloud instance is sufficient. It carries activations only on an explicit, last-resort fallback, and then only as ciphertext (more on this in Section IV-B); even then its cost is bandwidth rather than computation. The *host* creates the session and selects the model. Every other participant is a *member*, joining with a session code and reporting its measured capacity.

One member holds the *head*: the embedding, the unembedding, and the sampler, which are not divisible across shards. Which member holds it is decided by the planner rather than by who opened the session. The head's components are resident in addition to its blocks, so the choice of head and the division of blocks constrain each other, and Section V-C solves them together.

A session accepts prompts at every member, not only at the head, though only the head can actually answer one. Each member serves the same interface locally and forwards it, so a person uses the session from the device in front of them without knowing which machine is the head. This is the reciprocity the setting calls for. The laptop that contributes a shard is also where its owner types. It also means a session may be asked for several answers at once, by different people, which the planner's cache reservation must account for.

Membership defines the trust boundary. The session code is a bearer token, so any holder can join and will observe the activations that cross the shard it is assigned. This is the whole of the access-control model. A session is scoped to whoever the host shares the code with, which is the boundary a household, a laboratory, or a group of collaborators already maintains.

The members are arranged in a *ring*. A token passes from the head through each member's block range in turn and returns to the head. The return leg is what makes it a ring rather than a line: the head owns the unembedding, so the last member's output has to come back before a token can be produced. Within the ring, each member's compute overlaps the next member's transfer, as in prior pipeline-parallel systems. What a session adds is that the cost of an edge depends on whether its two members can reach each

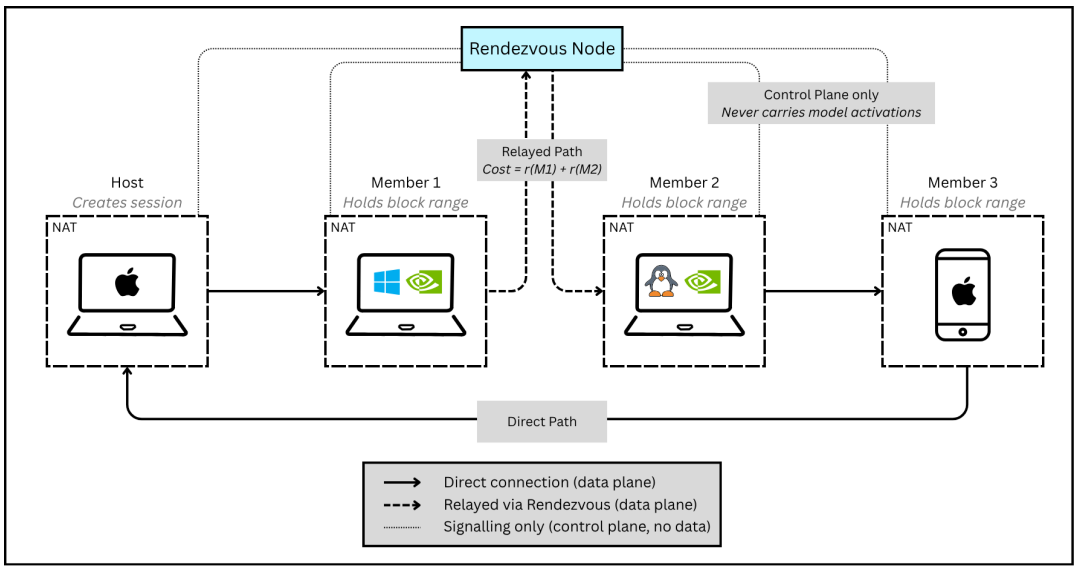

Fig. 1. A four-member session model, arranged as a ring from the host through three members and back.

other directly, which is not known before the session is formed. Choosing the ring order, and the block assignment within it, is the planner's responsibility.

A *session* model with four members is illustrated in Figure 1. Every member holds a long-lived signaling connection to the rendezvous, used only for registration and NAT-traversal coordination. The ring carrying the data plane is separate, and its edges are direct wherever traversal succeeds and relayed through the rendezvous only where it does not.

IV. Protocol: Forming a Session From Behind NAT

A naive approach has the host listen on a port and publish an address for members to connect to, and has every ring member listen for its predecessor. Both directions require accepting an unsolicited inbound connection, which a device behind NAT cannot do without manual router configuration. That assumption does not hold for most users. The protocol's governing constraint is therefore that no participant is ever required to accept an unsolicited inbound connection, or to configure a router. The rest of this section follows from that constraint.

A. Bootstrap and reachability discovery

The host opens a long-lived *outbound* connection to the rendezvous and registers a model identifier, an expected member count, and its own capability, receiving a short session code in return. A member connects outbound and presents the code. The rendezvous introduces the two, exchanging each side's candidate addresses.

Every member's reachability is classified before the ring is planned. Writing m for the number of members, probing every pair directly is an $O(m^2)$ network operation. It is also unnecessary. Whether a given pair can connect is largely a function of each node's own NAT behavior [20]: how it maps an internal port to an external one across different destinations, and whether it accepts a packet from a host it has not itself sent to. Both are measurable without involving the other party. Kafila therefore classifies each node once, against the rendezvous and STUN servers, at $O(m)$ probes; predicts every pair's feasibility from those measurements, which is $O(m^2)$ arithmetic and no packets; and then verifies only the edges the chosen plan uses, at $O(m)$ probes. The quadratic step is a table lookup, so network cost is linear in membership size at both ends.

Prediction is a prior, not a guarantee, and both directions of error are tolerated rather than corrected. An edge predicted direct that fails verification falls back to relaying, and the ring order computed from the prediction is kept. An edge predicted relayed that connects directly is cheaper than the plan assumed, and the session runs faster on it than predicted. Section VII-A reports how often each occurred against real equipment.

B. Traversal and relay fallback

The data plane runs over QUIC [22]. Using UDP avoids simultaneous-open TCP traversal, which succeeds less often than its UDP equivalent [17]. Kafila attempts a hole-punched direct path for every edge first, and relays through the rendezvous only where that fails.

The preference order is first a latency decision. A direct edge costs the round trip between its two members. A relayed edge costs the sum of both members' round trips to the rendezvous, because every frame travels to it and back out again. A ring pays that difference on every edge it relays, for every token of every request, so it compounds rather than being amortized. The planner prices the two cases in the same unit and orders the ring on the result.

Confidentiality does not turn on the path. The rendezvous mediates signaling, so it knows the session's membership and the members' addresses, but each edge's QUIC session is

encrypted end-to-end by TLS 1.3 [23], [24], whose keys the rendezvous never holds. Where an edge relays, the rendezvous forwards the members' encrypted datagrams rather than terminating their session, and so sees only ciphertext: it learns how much two members exchange, never the tensors they carry, and a direct edge keeps even that off it. Preferring direct paths narrows what the rendezvous observes to the signaling it already has, and removes the dependency on it, rather than protecting content the encryption protects on every path. Section VII-A reports the fraction of edges that stayed direct on real consumer connections.

V. Planner: Cost, Placement, Ordering, and Admission

Decoding at batch size one, producing one token at a time for a single in-flight request, is memory-bandwidth-bound [25], and a ring incurs its network cost on every token: a request's per-token circuit time is

$$T_{\text{circuit}} = \sum_{i} c_i + \sum_{(i,j)\in \text{ring}} \ell_{ij}, \tag{1}$$

the sum of each member's compute stage c_i and every ring edge's latency ℓ_{ij}. Two properties of (1) pull in opposite directions. The compute term is invariant to ring size: splitting a fixed forward pass across more members subdivides the same total work rather than adding to it. The latency term grows linearly with ring size, because each additional member introduces another hop paid on every token. The planner's task is to choose the block assignment and ring order that minimize (1) for a given membership, and to determine when no choice is feasible.

A. Cost model

Each stage's compute cost c_i is derived rather than fitted, so that it can be checked against measurement rather than tuned to it. At batch size one a stage reads every weight it holds exactly once, and performs two floating-point operations per weight, so memory bandwidth rather than arithmetic throughput dominates. This gives

$$c_i = o_i + n_i \cdot \frac{\beta_{\text{block}}}{\bar{b}_i} + \mathbb{1}[i = \text{head}] \cdot \frac{\beta_{\text{head}}}{\bar{b}_i}, \tag{2}$$

where $\bar{b}_i$ is device i's measured memory bandwidth and n_i the number of blocks it is assigned. The byte sizes β_{block} and β_{head} are properties of the model, identical for every device: one transformer block, and the head's embedding and unembedding matrices. The term o_i is a fixed per-token dispatch overhead, measured independently of role, and $\mathbb{1}[i = \text{head}]$ is 1 for the head and 0 otherwise. Every term is measured before a weight is loaded. A short device-side probe supplies $\bar{b}_i$ and o_i; the checkpoint's manifest supplies the two byte sizes. No term is fitted to an observed stage time, so the model can be tested against stage times it never saw.

Two things follow for head placement, and they are different. The first is cost, since the head pays $\beta_{\text{head}}/\bar{b}_i$ on every token, so where that lands matters, though only to the extent that devices differ in o_i as well, since every other term scales with bandwidth alike. The second is capacity, since the head's embedding and unembedding are resident alongside its blocks, so a device holding the head can hold fewer blocks than the same device could otherwise. The first makes head placement worth optimizing. The second makes it a constraint, and Section V-C solves the two together.

B. Ring ordering

Ordering is chosen over a connectivity graph that is *incomplete in addition to being unequal*: an edge may be direct, relayed at several times the cost, or, before verification, merely predicted either way. The planner's first pass is a closed-form heuristic over each member's NAT behavior class. Following RFC 4787 [20], members are permissive, meaning endpoint-independent in both mapping and filtering; restrictive, meaning endpoint-dependent mapping; or ordinary in between. A ring edge can be direct only if at least one endpoint is permissive, so an all-direct ring exists if and only if the permissive members are at least as many as the restrictive ones, and alternating the two around the anchor achieves one whenever it is achievable. This is a counting argument rather than a search, and stays linear in membership size.

Counting relayed edges is the right objective only where direct edges are uniformly cheap and relayed ones uniformly expensive, and neither holds once a session spans more than one site. On our testbed members measured median round trips of 79 to 115 ms to the rendezvous, so a relayed edge cost between 158 and 230 ms depending on which two members it joined. A rule that counts treats all of them as one quantity.

Where every member has measured its own round-trip time to the rendezvous, the planner replaces counting with a model that prices both kinds of edge in the same unit. Writing r_X for member X's measured round trip, a relayed edge $A \leftrightarrow B$ costs $r_A + r_B$, which is not an estimate but the literal path through the rendezvous. A direct edge is bounded below by the triangle inequality at $|r_A - r_B|$. Ring cost is the sum of both kinds around the cycle. The planner searches orderings exhaustively for the cheapest, holding one member's position fixed to remove the ring's rotational symmetry.

One case is degenerate. When every edge in a ring must relay, $\sum_{\text{edges}}(r_A + r_B) = 2\sum_{\text{members}} r_i$ for *any* arrangement of the same members, so every ordering scores identically and the model is silent exactly where it is needed most: it can place two members who share a network at opposite ends of the ring. Kafila breaks such ties on ring *separation*, the same $|r_A - r_B|$ term summed around the cycle, which prefers orderings that place members of similar rendezvous distance next to each other. A predicted-relayed edge sometimes connects anyway, which Section VII-A observes in two of nine class pairs and on the intercontinental fleet, and the arrangement that gains when it does is the one whose near members are already neighbors; where the prediction holds instead, adjacency costs nothing. This does not resolve the model's blindness between two equally relay-bound arrangements, but it stops the model choosing the worse of them.

C. Block placement

Fixing the ring order first turns block placement into a *chain-on-chain* partition: a contiguous chain of transformer blocks divided across processors of known, unequal speed, in a fixed processor order. Pinar et al. [16] show this is solvable exactly in polynomial time, in contrast to the NP-complete problem of choosing order and placement jointly.

The planner solves it with a parametric search, given as Algorithm 1 (The PARTITION algorithm). The search is stated over two per-device functions derived from (2): $\mathrm{fixed}(i) = o_i$, plus $\beta_{\mathrm{head}}/\bar{b}_i$ if i is the head, and $\mathrm{per}(i) = \beta_{\mathrm{block}}/\bar{b}_i$. A third function, $\lim(i)$, is device i's block limit at the session's declared context and concurrency. It is smaller when i is the head, because the head's resident state includes the embedding, and this is where head placement enters as a constraint rather than as a cost.

A device's stage cost is affine in the number of blocks it holds, so the bottleneck of any feasible division equals $\mathrm{fixed}(i) + n \cdot \mathrm{per}(i)$ for some device i and some integer n. The optimum is therefore one of at most $O(k \cdot m)$ candidate values, for k blocks and m devices, rather than an arbitrary real number (Algorithm 1, line 1). The search sorts these candidates and binary-searches them for the smallest that is *feasible* (line 2).

The FEASIBLE subroutine tests a candidate bound in one left-to-right pass, assigning each device the largest block count it can hold within the bound, subject to $\lim(i)$ and to reserving one block for every device still to come. The pass is optimal by an exchange argument: the chain is contiguous and the order fixed, so any block a device declines falls to a later one and cannot lower the bottleneck. Feasibility is monotone in the bound, so binary search over the sorted candidates is correct. Generating and sorting them dominates, giving $O(km \log km)$ overall. PARTITION returns $\tau^\star$ alongside the division, since head placement compares rotations by achieved cost.

Head placement reuses PARTITION rather than reopening the NP-complete joint problem, by the same rotation argument that fixes the ordering. A ring has no distinguished first device, so rotating the device sequence leaves every edge where it was, and the ordering already chosen is unaffected by which device serves as head. The planner therefore solves the fixed-order partition once per rotation, m exact solves rather than a search over $m!$ orderings, and keeps the rotation with the lowest achieved bottleneck, un-rotating its block counts to the original order.

Two devices of similar bandwidth are close to indifferent as head, since every non-overhead term in (2) scales with bandwidth alike. The choice then rests on the overhead terms o_i, and those are noisy. The two lowest-overhead devices on our testbed differed in median o_i by $1.5\,\mu$s, while one of the two varied by $8.4\,\mu$s across the same sessions. A difference smaller than one device's own spread would otherwise be enough to move the head. The planner therefore keeps the anchor as head unless some rotation improves on it by more than a margin, accepting the alternative only when $\tau^\star < \tau_{\mathrm{anchor}}(1-\mu)$ with $\mu = 0.05$, chosen against the noise it absorbs rather than against any particular gain. Overhead accounts for at most 2.2 per cent of a measured stage across every device and division we ran, and its run-to-run variation moves a stage by at most 1.1 per cent, so an apparent five per cent improvement is larger than overhead noise can produce.

Algorithm 1 PARTITION: exact chain-on-chain block placement for a fixed device order.

Require: k blocks; devices $1..m$ in ring order, device i costing $\mathrm{fixed}(i) + n \cdot \mathrm{per}(i)$ for n blocks, up to a limit $\lim(i)$

Ensure: block counts $n[1..m]$ and their bottleneck $\tau^\star$, or INFEASIBLE

1: $C \leftarrow \{\, \mathrm{fixed}(i) + n \cdot \mathrm{per}(i) : 1 \le i \le m,\ 1 \le n \le \min(k, \lim(i)) \,\}$, sorted ascending
2: $\tau^\star \leftarrow$ smallest $\tau \in C$ with FEASIBLE$(k, \tau) \neq$ INFEASIBLE {binary search over C}
3: **if** no such $\tau^\star$ **then**
4: **return** INFEASIBLE
5: **end if**
6: **return** FEASIBLE$(k, \tau^\star)$, $\tau^\star$
7: **function** FEASIBLE(k, τ):
8: left $\leftarrow k$
9: **for** $i = 1$ **to** m **do**
10: **if** $\mathrm{fixed}(i) > \tau$ **then**
11: **return** INFEASIBLE
12: **end if**
13: $n[i] \leftarrow \min\big(\lfloor(\tau - \mathrm{fixed}(i))/\mathrm{per}(i)\rfloor,\ \lim(i),\ \mathrm{left} - (m - i)\big)$
14: **if** $n[i] < 1$ **then**
15: **return** INFEASIBLE
16: **end if**
17: left $\leftarrow$ left $- n[i]$
18: **end for**
19: **if** left $\neq 0$ **then**
20: **return** INFEASIBLE
21: **end if**
22: **return** n

D. Admission

Ordering and placement both depend on inputs a device may not have. A device that failed its bandwidth probe cannot be priced by (2), and one that failed to time itself against the rendezvous cannot be ordered by measured cost. Refusing to plan whenever an input is missing is not an option, since a session is only as well instrumented as its worst-measured member. The planner instead degrades through three tiers, each better informed than the one below.

If every member priced itself under the full cost model, the search of Section V-C runs as described. If some could not, but every member reported raw arithmetic throughput, the same search runs with $\mathrm{per}(i)$ set from that ratio and no head term, since arithmetic throughput does not price the head's extra read; the head then stays where the session was opened. Only the lowest tier abandons the search: with nothing measured,

blocks are divided in proportion to reported working set, which is the least informed division and the one both tiers above improve on.

Admission is checked once, after a division has been produced by whichever tier applies, against every member's available room at the session's declared context and concurrency. If the assignment exceeds what a device can hold, the plan is rejected and the undersized device is named, rather than reporting only that the session cannot run. When the full cost model has already tried every division that minimizes the bottleneck and none fits, the only remaining reason to refuse is memory. The planner refuses, rather than return a plan that would exhaust a device partway through serving. This is a feasibility criterion rather than a usability one. It establishes that a division exists within the members' collective memory, not that the resulting circuit time is worth serving.

VI. Implementation and Experimental Setup

Kafila is implemented as a fork of Ollama [26], a widely used local-inference server, retargeted to MLX [27], whose Metal and CUDA backends let the same session and planner code run unmodified across Apple silicon and Nvidia hardware. A session's membership mixes consumer platforms, so no code path is specific to a platform or a region. Weights are fetched per shard from the model host's safetensors files by byte range, so a member holding one-tenth of a model downloads one tenth of it. Shards of the same checkpoint share tensors, so replanning a division after a partial fetch costs little.

The rendezvous builds without the inference runtime, since it holds no model and needs no accelerator, and runs as an unprivileged service on a two-core cloud instance. Every session records predicted and measured cost, circuit time, and bytes transferred, per stage and per hop, and each trace separates time-to-first-token into prefill (processing the prompt) and decode. Each cell is five generations, the first discarded as cold, and reported data is the median of the rest. All the data in this paper is captured using that instrumentation.

A. Testbed and NAT emulation

The three fleets hold the members roughly fixed and vary how far apart they sit (Table II), so that the network overhead, the fraction of each token spent on the network rather than computing, runs from 12 per cent on the shared LAN to 94 across continents. Every fleet mixes consumer graphics cards, consumer-sized slices of larger cards, an institutional GPU, and an Apple laptop, and on every one speed and memory rank differently: the LAN fleet's A40 holds the least memory yet reads faster than the laptop, and the laptop holds middling memory while reading an order of magnitude slower than the fastest card. No share assigned by memory alone matches what a device can serve. Across the fleets bandwidth spans an $11\times$ range and memory a $4\times$ range. Home uplinks are not reproduced, which affects fetch time rather than the served rate we report, and intermittency and churn are outside this paper's scope.

TABLE II
The three evaluation fleets

Fleet	Device	Mem	Bw (GB/s)	Site
LAN	L40S	24	512–529	Melbourne, AU
(4 dev)	L40	24	367–453	Melbourne, AU
	A40	12	377–439	Melbourne, AU
	M3 Pro	18	89–124	Melbourne, AU
US Central	A100	80	915–959	Des Moines, US
(5 dev)	RTX 5090	32	849–1047	Chicago, US
	RTX 3090	24	673–714	Marion, US
	RTX A6000	48	393–436	Kansas City, US
	RTX A6000	48	400–480	Kansas City, US
Inter-	RTX 5090	32	1303–1405	Arad, RO
continental	A100	40	812–949	Prague, CZ
(5 dev)	RTX 3090	24	625–704	Teplice, CZ
	L40S	24	505–526	Melbourne, AU
	M3 Pro	18	89–127	Melbourne, AU

The three models served are Qwen3-8b, Qwen3-14b and Qwen3-32b [4], each generation decoded greedily at a fixed seed so allocations are compared on identical output. The class pairs that bear most on the ordering rule are the ones a real network is least likely to offer, so those results are obtained under Tailscale's `natlab` [28], which models the RFC 4787 matrix directly. Only the NATs are emulated: the candidate exchange, the hole punch, the QUIC handshake, the reachability classifier, and the rendezvous carrying signaling are the deployed implementation, running unmodified over emulated links.

VII. Performance Evaluation

The protocol and the planner rest on different instruments, so we evaluate them apart. The protocol's claim is about NAT behavior, which we exercise on real consumer connections and, for the classes a real network seldom offers, under emulation. The planner's claim is about heterogeneous compute, which needs unequal real machines rather than a simulated cost model, so it is measured on the three fleets of Table II.

We compare Kafila against two existing algorithms [11], [12]. Uniform gives every member the same number of blocks, which is GPipe's rule [11]. Memory-proportional gives each a share proportional to its working set, exo's default for rings of personal devices [12]. PipeEdge [13] and PipePar [14] price a stage from a profile taken with the weights already resident, which a session cannot obtain before it has decided who holds what. Each cell is five independent sessions; the network is drawn afresh in each, so throughput carries its five-session half-range while the compute terms, taken on each device's own clock, do not.

A. Forming a session across real NATs

Real networks separate the cases the emulator cannot. On the US Central fleet the five members sat behind five different public networks, and traversal punched every ring edge direct over public addresses at a median of 1.29 s, so no activation ever left the served ring. The four LAN members instead sat on one private network, and their edges connected over local addresses at 1.26 s; the reachability rule read all four as restrictive

TABLE III
NAT CLASS-PAIR TRAVERSAL OUTCOMES

Initiator	Responder	Rule predicts	Traversal achieves
permissive	permissive	direct	direct
permissive	ordinary	direct	direct
permissive	restrictive	direct	direct
ordinary	permissive	direct	direct
ordinary	ordinary	direct	direct
ordinary	restrictive	relay	direct
restrictive	permissive	direct	direct
restrictive	ordinary	relay	direct
restrictive	restrictive	relay	relay

and predicted relays, but the verdict cost nothing because no edge needed the rendezvous to begin with. Across continents the fallback carries the load: on the intercontinental fleet only the two co-located members reached each other directly, at 2.24 s, and the other four ring edges relayed at a median of 20.8 s, so roughly four fifths of each token's activation bytes crossed the rendezvous as ciphertext. A ring planned around a relay that proves unnecessary costs nothing at run time, whereas the reverse would strand it on a path that does not exist, so the rule errs only in the safe direction. In every session the ordering search returned an arrangement optimal in predicted relayed edges, verified against an exhaustive check.

Table III emulates all nine ordered pairs of NAT behavior class against the deployed traversal implementation. The reachability rule's prediction matched the outcome in seven pairs, shown green, and mispredicted two, shown red; both errors forecast a relay where traversal in fact succeeded directly, never the reverse. Both errors are the ordinary-against-restrictive pairing, where an endpoint-independent mapping lets the punch answer the source a packet arrived from rather than the address that was advertised.

B. Dividing the model

Kafila divides against measured speed, and the effect on the division is large and holds on every fleet (Table IV). Its slowest stage exceeds its fastest by 1.4 to $1.8\times$, where a memory-proportional split reaches 2.1 to 5.5 and a uniform one up to 10.6. The heuristics do not spread the excess evenly: each buries the laptop, which holds middling memory and reads an order of magnitude slower than the fastest card, so a share sized by memory is work it cannot get through, and in a ring every token waits for it. Kafila's bottleneck stage is consequently up to $5.2\times$ shorter than uniform division and up to $3.0\times$ shorter than a memory-proportional one. Read the other way this is utilisation, the average stage's compute over the bottleneck's: the share of committed device-time that computes rather than waiting on the slowest stage, which Kafila holds at 75 to 87 per cent where the heuristics fall below half. It is not the imbalance column renamed, which weighs only the fastest and slowest stages; utilisation counts every one, and it sets the ceiling the multi-user mode reaches: the four-user throughput gain climbs with it, from near $2\times$ at the heuristics' 42 per cent to near $4\times$ at Kafila's 86

TABLE IV
KAFILA AGAINST BASELINES, BEST BOLD

Model	Allocation	Imbal. max/min	Bneck ms	Util.	tok/s
LAN, 12% network overhead					
Qwen3-8b	Kafila	**1.46**	**11.3**	**86%**	**21.5** ±0.2
	Mem-prop. (exo)	4.01	26.0	48%	17.2 ±3.8
	Uniform (GPipe)	5.14	36.2	41%	13.7 ±1.3
US Central, 38 to 62% network overhead					
Qwen3-8b	Kafila	**1.62**	**5.2**	**81%**	**17.4** ±1.2
	Mem-prop. (exo)	2.10	5.6	77%	15.0 ±1.4
	Uniform (GPipe)	1.66	5.6	81%	16.8 ±2.1
Qwen3-14b	Kafila	**1.46**	**8.1**	**82%**	**12.8** ±0.8
	Mem-prop. (exo)	2.50	9.7	72%	12.4 ±2.3
	Uniform (GPipe)	2.07	9.6	75%	11.1 ±1.2
Qwen3-32b	Kafila	**1.37**	**17.4**	**80%**	8.8 ±1.4
	Mem-prop. (exo)	2.83	21.2	69%	**8.9** ±1.3
	Uniform (GPipe)	2.28	20.8	74%	8.1 ±1.1
Intercontinental, 84 to 94% network overhead					
Qwen3-8b	Kafila	**1.84**	**6.2**	**75%**	**2.3** ±0.2
	Mem-prop. (exo)	5.38	15.6	41%	2.3 ±0.3
	Uniform (GPipe)	10.11	24.9	33%	2.3 ±0.2
Qwen3-14b	Kafila	**1.51**	**8.3**	**87%**	**2.3** ±0.2
	Mem-prop. (exo)	4.63	23.5	43%	2.2 ±0.2
	Uniform (GPipe)	10.59	43.5	32%	2.1 ±0.1
Qwen3-32b	Kafila	**1.46**	**18.1**	**85%**	**1.9** ±0.4
	Mem-prop. (exo)	5.49	54.6	41%	1.9 ±0.1
	Uniform (GPipe)		*infeasible*		

(Section VII-C). These are per-device times on each device's own clock, independent of the network between them.

Getting the division right can decide whether a model runs at all. On the intercontinental fleet no uniform split of Qwen3-32b fits the 18 GB laptop's equal share once cache is reserved for four concurrent requests, so no session forms; the planner assigns an uneven division, names the device a uniform split would overload, and serves the model. The same memory wall bounds the LAN fleet's 12 GB member on Qwen3-14b, so that fleet is evaluated on Qwen3-8b.

What the equalized compute returns to a user depends on how much of a token is computation rather than network, the overhead the fleets vary from 12 per cent to 94. Where members share a LAN the planner's division delivers $1.56\times$ the throughput of a uniform split and $1.25\times$ of a memory-proportional one, with non-overlapping five-session ranges. The closer the members sit, the more of the planner's compute advantage the user receives: as the network share rises the same advantage occupies less of each token, so on the intercontinental fleet the allocations serve at rates within the five-session spread of one another even as their bottleneck and utilisation stay as far apart as the table shows. The planner governs the compute term, and a group whose devices share a network collects that term in full.

The planner makes three decisions from one cost model, to different ends: the division sets throughput, head placement keeps the plan within memory, and the ring order holds network cost down. The throughput is the division's, and the cleanest ablation of it is the headline: dividing evenly or by

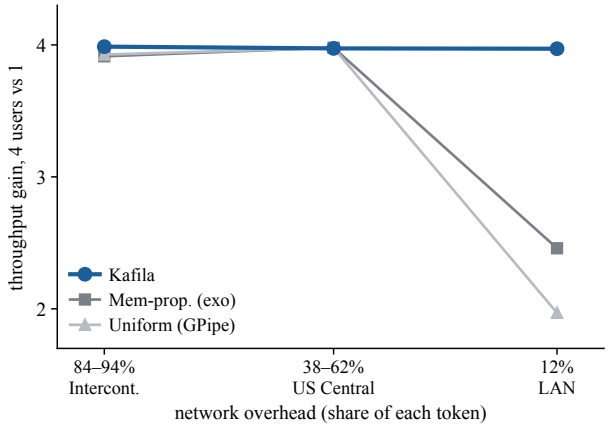

Fig. 2. Four-user throughput gain over one user against network overhead. Only Kafila's balanced division scales near-linearly at every overhead; the imbalanced divisions drop away as the overhead falls. Points are per-fleet medians across models.

memory, without the cost model, is the uniform and memory-proportional split and forfeits the $5.2\times$ and $3.0\times$ of Table IV. Pinning the head where the session opened, and ordering by relay count rather than cost, then leave the served division, its bottleneck, and its throughput within run-to-run variance on every fleet. That is expected, since neither is a throughput knob and these fleets do not stress it: the opener could hold the head, and every LAN and US edge went direct. Their value is elsewhere, in a head that will not fit declining a session and a cost-blind order relaying where it need not. The cost model is fitted to nothing, yet predicts per-stage compute to a median of 18 per cent across 165 stages, so the residual imbalance is that prediction gap, not a limit of the method.

C. Serving several users at once

Sequential decoding leaves a bubble at every stage: a token must pass through all k members before the next begins, so at any instant one stage computes and the rest wait. Concurrent requests fill that bubble, letting each stage work on one user's token while another stage works on the next user's. The bubble is there at any network share; a slow network only widens it. Figure 2 reports four concurrent users against one on all three fleets; to check that the batch-one allocation is still the right one under load, we run every division, not only the planner's.

The LAN fleet is the decisive case, because at 12 per cent network overhead there is almost no network idle to hide behind. There the planner's division raises total throughput by $4.0\times$ while the memory-proportional and uniform divisions reach only 2.5 and $2.0\times$. The imbalanced divisions stall at their overloaded stage, which was already near capacity with one user, so extra requests queue behind it rather than filling other members. Only the balanced division has the per-stage headroom to absorb four requests, so the planner's single-user throughput lead of $1.6\times$ over uniform division widens to $3.2\times$ under four users. This answers the concern that a bottleneck objective chosen for one request might be wrong for many. Batching four requests raises arithmetic intensity, but the member an imbalanced split overloads is the slowest on both axes that shift, streaming weights and computing, so it stays the binding stage as the regime moves; the division that unburdens it for one request is the one that scales to four. Were the objective wrong for multi-user mode the lead would shrink under load; on the LAN it doubles.

Where the network is slower it supplies the idle that a poor division would otherwise lack, so every division fills and the gains converge. On US Central all three reach 3.9 to $4.0\times$ across the three models while each user's rate changes by 1.0 per cent or less, and the planner leads on absolute throughput at every size but the largest, where it ties. On the intercontinental fleet, where network overhead is 84 to 94 per cent, throughput scales 3.5 to $4.0\times$ for all divisions and, as with a single user at that distance, they fall within session spread: the network masks the division. Serving four users is therefore near-free wherever a session is worth forming, and where the members share a network the planner both serves one user faster and admits more of them before its smallest device runs out of cache.

VIII. Conclusions and Future Work

A trusted set of heterogeneous, commodity machines can serve an LLM that none of them could run alone. A session must form despite the NATs its members sit behind, and be worth using once formed. This paper proposed a protocol for the first and a planner for the second, measured across three fleets that vary how far apart the members sit. Every admitted session formed without router configuration. Against the two divisions established in prior work, the planner shortens the slowest stage by up to $5.2\times$ relative to uniform division and up to $3.0\times$ relative to a memory-proportional one, keeps 75 to 87 per cent of the committed hardware working where those divisions fall below half, and serves a model uniform division cannot place at all. Its cost model is fitted to nothing, yet predicts per-stage compute to a median of 18 per cent. How much of that advantage a user sees depends on how much of a token is computation rather than network overhead, which we measured from 12 to 94 per cent: where members share a network the same division returns $1.56\times$ the throughput of uniform division and $1.25\times$ of a memory-proportional one, and devices in one lab or home sit at that end. Under four concurrent users that lead does not fade but compounds to $3.2\times$, because a fast network leaves little idle for an imbalanced division to hide behind, so the balance Kafila's planner strikes is what lets a session scale to many users rather than merely serve one well.

Four directions remain for future work. The largest is the network term, and these measurements bound what reducing it could recover. Membership that changes mid-session needs cache recovery around a replan the planner can already compute. A mobile member changes exactly the two properties the planner conditions on, so the question is how often to replan rather than how. And cache reserved for prompts that never arrive would return to the model if shared across members. A session of invited devices gives up the redundancy that lets a swarm route around its weakest participant. What it

offers in exchange is a group whose membership, capability and connectivity are known before serving begins, and which divides well enough to serve models no member could hold alone.

REFERENCES

[1] F. Dehghani, R. Dehghani, Y. Naderzadeh Ardebili, and S. Rahnamayan, "Large language models in legal systems: A survey," *Humanities and Social Sciences Communications*, vol. 12, 2025.

[2] K. He, R. Mao, Q. Lin, Y. Ruan, X. Lan, M. Feng, and E. Cambria, "A survey of large language models for healthcare: From data, technology, and applications to accountability and ethics," *Information Fusion*, vol. 118, p. 102963, 2025.

[3] Menlo Ventures, "2025 mid-year LLM market update: Foundation model landscape and economics," https://menlovc.com/perspective/2025-mid-year-llm-market-update/, 2025.

[4] Qwen Team, "Qwen3 technical report," 2025.

[5] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-throughput generative inference of large language models with a single GPU," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[6] A. Borzunov, D. Baranchuk, T. Dettmers, M. Ryabinin, Y. Belkada, A. Chumachenko, P. Samygin, and C. Raffel, "Petals: Collaborative inference and fine-tuning of large models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL): System Demonstrations*, 2023, pp. 558–568.

[7] M. Ryabinin, T. Dettmers, M. Diskin, and A. Borzunov, "SWARM parallelism: Training large models can be surprisingly communication-efficient," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[8] C. Tong, Y. Jiang, G. Chen, T. Zhao, S. Lu, W. Qu, E. Yang, L. Ai, and B. Yuan, "Parallax: Efficient LLM inference service over decentralized environment," 2025.

[9] M. Fan, Y. Liu, F. Wang, and C. Chen, "What does the server see? understanding privacy leakage from large language models in split inference," 2026.

[10] E. Erdogan, A. Kupcu, and A. E. Cicek, "UnSplit: Data-oblivious model inversion, model stealing, and label inference attacks against split learning," in *Proceedings of the 21st Workshop on Privacy in the Electronic Society (WPES)*, 2022.

[11] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019.

[12] exo Labs, "exo: Run your own AI cluster at home with everyday devices," https://github.com/exo-explore/exo, 2025.

[13] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. Walters, "PipeEdge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices," in *Proceedings of the 25th Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 298–307.

[14] J. Zhang, G. Niu, Q. Dai, H. Li, Z. Wu, F. Dong, and Z. Wu, "PipePar: Enabling fast DNN pipeline parallel training in heterogeneous GPU clusters," *Neurocomputing*, vol. 555, p. 126661, 2023.

[15] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[16] A. Pinar, E. C. Tabak, and C. Aykanat, "One-dimensional partitioning for heterogeneous systems: Theory and practice," *Journal of Parallel and Distributed Computing*, vol. 68, no. 11, pp. 1473–1486, 2008.

[17] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC)*, 2005.

[18] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session traversal utilities for NAT (STUN)," RFC 5389, IETF, 2008.

[19] A. Keranen, C. Holmberg, and J. Rosenberg, "Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal," RFC 8445, IETF, 2018.

[20] F. Audet and C. Jennings, "Network address translation (NAT) behavioral requirements for unicast UDP," RFC 4787, IETF, 2007.

[21] D. Trautwein *et al.*, "Large-scale measurement of NAT traversal for the decentralized web: A case study of DCUtR in IPFS," 2026.

[22] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," RFC 9000, IETF, 2021.

[23] M. Thomson and S. Turner, "Using TLS to secure QUIC," RFC 9001, IETF, 2021.

[24] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," RFC 8446, IETF, 2018.

[25] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," in *Proceedings of Machine Learning and Systems (MLSys)*, 2023.

[26] Ollama Inc., "Ollama: Get up and running with large language models," https://github.com/ollama/ollama, 2024.

[27] A. Hannun, J. Digani, A. Katharopoulos, and R. Collobert, "MLX: Efficient and flexible machine learning on apple silicon," 2023. [Online]. Available: https://github.com/ml-explore

[28] Tailscale, "`tstest/natlab`: An in-process virtual network for testing NAT traversal," https://pkg.go.dev/tailscale.com/tstest/natlab, 2025, tailscale v1.102.3.