Distributed AI Gateway: A Fault-Tolerant, Routing Based Compute Cluster for Federated LLM Inference

Divya Rajasekar
Computer Engineering Department
San José State University
San José, California, United States
divya.rajasekar@sjsu.edu

Conlyn Pattison
Computer Engineering Department
San José State University
San José, California, United States
conlyn.pattison@sjsu.edu

Shivansh Chhabra
Computer Engineering Department
San José State University
San José, California, United States
shivansh.chhabra@sjsu.edu

Abhinand Vijayakumar Binsu
Computer Engineering Department
San José State University
San José, California, United States
abhinand.vijayakumarbinsu@sjsu.edu

Abstract *- Large language model (LLM) inference is compute-intensive and typically centralised, creating cost and availability bottlenecks for resource-constrained teams. This paper presents the Distributed AI Gateway, a leader to worker compute cluster that aggregates heterogeneous commodity laptops connected over a Tailscale VPN mesh to serve collaborative AI inference. The system routes user prompts from a React web application through a FastAPI leader node, which dispatches tasks via Apache Kafka to registered worker nodes running Ollama with locally available open-weight models. Leadership continuity is maintained through a heartbeat-driven failure detector and a convergent priority-based designation protocol backed by a cloud-hosted discovery service, enabling the cluster to survive leader loss without operator intervention. A two-phase, topic-level affinity routing scheme matches tasks to workers based on advertised RAM capacity, while Kafka consumer groups ensure single-consumer delivery within each tier. Apache Cassandra handles user authentication. Evaluation on a four-node prototype demonstrates sub-11-second leader handover, automatic worker task requeue, and correct ram tier routing across three RAM tiers. The architecture offers a practical, privacy-preserving alternative to cloud inference APIs for academic and small-team environments.*

Keywords *- distributed inference, leader election, Kafka, Ollama, Tailscale VPN, heartbeat failure detection, Ram Tier routing, federated compute*

I. Introduction

Agentic AI workloads and LLM-assisted development have made on-device inference a routine requirement. Cloud inference APIs eliminate hardware barriers but introduce per-token billing, mandatory data egress, vendor availability risk, and regulatory exposure for sensitive workloads. For academic teams, early-stage startups, and privacy-sensitive enterprises, these trade-offs are untenable.

At the same time, most engineering environments contain a latent pool of high-specification laptops that sit idle during meetings, seminars, and off-hours. A 64 GB developer workstation capable of running Llama-3 70B in full precision may be used for web browsing for the majority of its operational day. The central thesis of this work is that such heterogeneous, sporadically available hardware can be orchestrated into a coherent, fault-tolerant inference cluster without dedicated datacenter infrastructure.

The Distributed AI Gateway addresses this opportunity. Its design goals are: (1) zero-configuration node enrolment - any laptop on the VPN can register as a worker; (2) Ram-Tier task routing - tasks are matched to workers capable of hosting the required model; (3) transparent failover - leader and worker failures are detected via heartbeats and recovered without client-side configuration changes; and (4) privacy preservation - model inference runs entirely on-premises with no prompt data leaving the VPN.

This paper makes the following contributions. We describe a complete reference architecture for federated laptop-based LLM inference (§III). We formalise a RAM-capacity routing heuristic and a three-tier node failure taxonomy

(§III–IV). We demonstrate leader election with Cloud based dynamic re-routing and worker task requeue under simulated hardware failure (§IV). We characterise recovery latency on our four-node prototype (§VI).

II. Background and Related Work

A. Distributed Inference Scheduling

Prior work on distributed inference has focused on GPU cluster scheduling. vLLM [1] introduces paged attention to maximise GPU memory utilisation, while Orca [2] proposes iteration-level scheduling to reduce head-of-line blocking. Both assume homogeneous, always-on accelerator pools. Our setting differs fundamentally: nodes are heterogeneous commodity CPUs with intermittent availability, favouring a simple queue-pull model over complex preemption schedulers.

B. Leader Election and Failure Detection

The Raft consensus algorithm [3] provides a well-understood primitive for leader election in replicated logs. Our design borrows the heartbeat-timeout idiom from

these systems but opts for a convergent priority-based designation rather than a full Raft log, trading strong consistency for operational simplicity appropriate to a small cluster. The Phi Accrual Failure Detector [5] replaces binary alive/dead judgements with a continuous suspicion level φ, informing our understanding of the false-positive tradeoff in detection latency.

C. VPN-Overlay Peer Networking

Tailscale [6] implements a WireGuard-based mesh VPN with automatic NAT traversal, providing authenticated point-to-point connectivity without port-forwarding or public IP addresses. This overlay is the connectivity substrate our system exploits, analogous to RDMA fabrics in datacenter inference but adapted for consumer hardware operating behind residential NAT.

D. On-Device LLM Runtimes

Ollama [7] packages GGUF-quantised models behind a REST API, enabling single-command model loading on consumer hardware. Combined with llama.cpp's CPU inference path, it allows Mistral-7B to run on 8 GB of unified memory and Llama-3 70B on 64 GB, directly informing the RAM-tier routing thresholds described in §III-D.

III. System Architecture

Figure 1 provides a high-level view of the system. All nodes reside on a Tailscale VPN overlay. Client devices access the React frontend at port 3000, which communicates exclusively with the leader node's FastAPI endpoint on port 8000. The leader maintains a worker registry, publishes tasks to Apache Kafka, and exposes an admin dashboard. Workers pull tasks from Kafka, execute inference via Ollama, and publish results back. Apache Cassandra stores user accounts and audit logs.

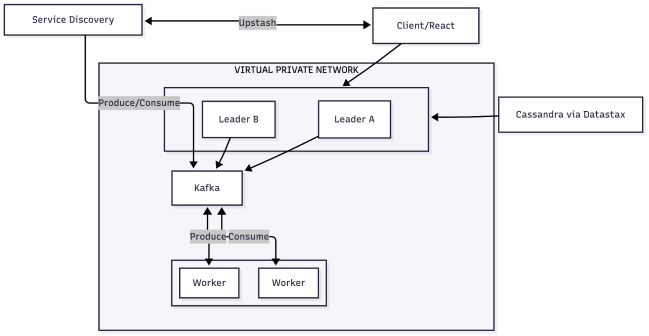

Fig. 1. System architecture. Client devices connect to the Leader over Tailscale VPN. The Leader dispatches tasks via Kafka to RAM-tiered Workers running Ollama.

A. Leader Node

The leader node exposes five functional REST endpoint groups. The *POST /register* interface accepts a worker's advertised metadata, encompassing a unique node identifier, advertised RAM capacity in gigabytes, the locally active Ollama model name, and a skill-set taxonomy derived from local configuration files. To mitigate spoofing, the system extracts the worker's IP address directly from the incoming HTTP connection substrate. Leadership continuity and membership lease renewal are facilitated via *POST /heartbeat*; missed heartbeats within a 15 s window trigger a DEAD status, whereupon Kafka's consumer-group protocol facilitates task requeue, making unacknowledged messages eligible for re-delivery to the next available eligible node. POST /ask accepts a user prompt, classifies it to an appropriate Kafka topic using the two-phase routing scheme (§III-D), and blocks on an asynchronous event until the result arrives from the Kafka reply consumer. POST /ask/stream provides the same flow as a Server-Sent Events (SSE) endpoint, delivering real-time status updates and the final response as a stream of typed events. GET /cluster/* exposes aggregate cluster health metrics, node topology, and request history to the admin dashboard.

B. Worker Nodes

The worker daemon is implemented as a singular Python process employing two concurrent threads to manage lifecycle and execution. The primary execution thread orchestrates a Kafka consumer loop with a 1 s poll timeout; upon task ingestion, it initiates a local HTTP request to the Ollama API, assembles the resultant completion, and dispatches a result message to the completed-tasks topic. To guarantee at-least-once delivery semantics, Kafka offsets are committed manually only following the verified publication of the result. Simultaneously, a background heartbeat thread facilitates membership lease renewal by transmitting a POST /heartbeat to the leader at 5s intervals, broadcasting node telemetry and cumulative throughput metrics. Should the heartbeat mechanism detect a leader failover—signaled either via a 503 response containing updated topology or through a discovery service lookup following consecutive failures it utilizes a thread-safe flag to trigger a seamless swap of the consumer and producer broker connections. This decoupled architecture ensures that the web-based administrative session and the underlying worker daemon remain state-independent, permitting a singular hardware node to concurrently function as both a cluster worker and an admin client.

C. Messaging via Apache Kafka

Tasks are serialised as JSON messages across three tiered topics: tasks-high-ram, tasks-low-ram, and tasks-general. Each message carries a correlation ID (request_id), the prompt text, user identity, and a timestamp. Workers subscribe to topic subsets based on their RAM capacity: nodes with ≥16 GB subscribe to all three topics; nodes with ≥8 GB subscribe to tasks-low-ram and tasks-general; all workers subscribe to tasks-general regardless of RAM capacity. All workers share the consumer group workers, so Kafka distributes each task to exactly one worker within the subscribed tier. Results are published to a fourth topic, completed-tasks, consumed by a background thread in the leader that resolves the waiting asynchronous event for the corresponding request_id.

The Kafka deployment uses KRaft mode (Kafka Raft metadata). In production, each leader-eligible laptop runs a co-located KRaft broker, forming a three-broker cluster with a replication factor of 3 and min.insync.replicas=2, ensuring the message log survives any single-broker failure.

D. Routing

Prompt routing operates across two independent dimensions: capacity tier and domain skill. The leader resolves both dimensions before publishing a single task to Kafka.

Tier Classification: The leader estimates generation cost using a token-count heuristic, computed as character count

divided by four. Prompts exceeding estimated tokens are assigned to the tasks-high-ram tier, those above 500 tokens to tasks-low-ram, and all remaining prompts to tasks-general. Each tier maps to a dedicated Kafka topic. This classifier is modular and can be extended to incorporate model-metadata lookups without modifying the underlying topic-level contract.

Skill Classification: Independently, the leader scans prompts for domain-specific keywords—such as coding, summarization, or creative-writing terms—to facilitate RAM Tier routing steering. If a skill is detected, the corresponding metadata is attached to the Kafka message.

Subscription-Based-Balancing: Workers subscribe to tiered topics based on advertised RAM capacity: nodes with ≥16 GB ingest messages across all tiers, while nodes with ≥8 GB are limited to tasks-low-ram and tasks-general. Kafka's consumer-group protocol facilitates single-consumer delivery within each subscribed tier, effectively decoupling the leader's routing heuristics from cluster-wide load distribution.

E. Authentication and Data Layer

Cassandra stores three tables in the web_app keyspace. The users table (partition key: user_id) stores credentials, display names, and role assignments. The requests table (partition key: user_id, clustering key: created_at DESC) stores per-user chat history. The cluster_requests_by_month table (partition key: time_bucket, clustering key: created_at DESC) provides an admin-facing view of all cluster activity, partitioned by calendar month to bound partition size — a Cassandra best practice for time-series data.

Three roles — admin, client, and server — provide distinct privilege tiers. Admins access the full dashboard with user management and cluster-wide request history. Clients access the chat interface. Server-role users see a node-status view with live activity logging.

IV. Failure Handling and Leader Election

A. Worker Failure - Task Requeue

Scenario: a worker pulls a task and its laptop is shut before completing inference. The leader detects the missing heartbeat after 15 s and marks the worker DEAD. Kafka's consumer session timeout (30 s) eventually expires without an acknowledgement, making the task eligible for re-delivery. The next available eligible worker picks it up. The user observes additional latency (typically 30–60 s on our prototype) but receives a correct result; no data is lost.

B. Leader Failure - Election Protocol

Each gateway-eligible node maintains a continuous leader process instance, though request-handling remains dormant on non-elected nodes. Peer nodes execute a background *LeaderMonitor* task, interrogating the */health* endpoint at 5 s intervals. Upon the detection of three consecutive failures, a node designates the incumbent leader as unreachable and initiates the election protocol.

The election mechanism is convergent and deterministic: each node independently computes a ranking of all active gateway-eligible peers based on configured priority, with node ID serving as a tie-breaker. By utilizing an eventually-consistent membership state—propagated through *POST /cluster/sync* and persisted via local YAML state files—cluster nodes converge on a singular winner without requiring explicit message exchange.

Upon designation, the newly elected node activates its request-handling substrate, broadcasts the updated topology to the cluster majority via *POST /cluster/sync*, and registers its endpoint with the cloud-hosted discovery service (Upstash Redis). This sequence ensures immediate reachability for both workers and client applications.

This architectural choice prioritizes operational simplicity over strong consistency, a trade-off appropriate for small-scale deployments. For clusters of 3–5 nodes, the deterministic priority-based designation scheme provides sufficient robustness while avoiding the disproportionate overhead of a full Raft log implementation.

The dormant leader paradigm functions as a core architectural primitive, ensuring seamless governance transitions.

C. Network Partition - Split-Brain Prevention

Network partitions present a significant risk of split-brain behavior by isolating the incumbent leader from the cluster majority. To mitigate this, the system implements an active quorum verification mechanism executed on every 5s monitor tick. The acting leader pings the */health* endpoints of all leader-eligible peers to calculate the current reachability count. Defining *reachable_peers* as the set of accessible nodes excluding the leader itself, the system computes the aggregate availability as:

$$available = reachable\_peers + 1$$

where the additional unit represents the leader's own state. If the condition:

$$available < floor(total\_eligible / 2) + 1$$

is met, the leader concludes it no longer maintains a majority quorum and initiates a self-demotion protocol. During this transition, the node ceases publication to the cloud discovery service and activates a request gate. This gate intercepts user-facing endpoints—including */ask*, */register*, */heartbeat*, */auth/*, and */admin/*—returning an HTTP 503 response populated with an *X-Leader-URL* header. Workers receiving this signal immediately re-resolve the cluster topology via the discovery service and migrate their registration to the majority-elected leader.

This threshold-based confinement prevents a minority-side leader from accepting novel inference prompts or managing worker state, effectively neutralizing stale service within the isolated partition. Concurrently, nodes within the majority partition utilize the heartbeat-failure threshold to detect the former leader's absence and execute the convergent priority-based election protocol to restore cluster governance.

Upon the restoration of network connectivity, the demoted node's monitor task observes that a quorum is once again reachable. The node then resumes discovery service publication and re-enables standard request handling. Peer nodes synchronize their membership state through *POST /cluster/sync* to converge on the single highest-priority active node. Given that LLM inference is effectively idempotent from a client perspective, any tasks independently processed during the partition interval do not introduce state-level consistency conflicts.

V. Implementation

The system is implemented across four components in a mono-repository with Docker Compose orchestration. Table I summarises the technology stack.

TABLE I. TECHNOLOGY STACK

Component	Technology	Language / Version
Frontend	React (CRA)	JavaScript / Node 20
Leader API	FastAPI + Uvicorn	Python 3.11
Task Queue	Apache Kafka	CP-Kafka 7.5 (KRaft)
Worker Runtime	Ollama + llama.cpp	Go / C++ (prebuilt)
Data Store	Apache Cassandra	Cassandra 5
Networking	Tailscale WireGuard	—
Discovery	Upstash Redis	REST Redis
Container	Docker Compose	Docker 25

A. Leader Node

The FastAPI application consolidates all service endpoints within a singular application instance, structurally partitioned into distinct modules for cluster coordination (encompassing registration, heartbeats, and membership synchronization), inference services (supporting both synchronous /ask and asynchronous SSE /ask/stream flows), user authentication (login and signup), and comprehensive administrative oversight (including user governance and historical request telemetry).Worker state is held in an in-memory registry with an asynchronous timeout loop. Cluster membership is additionally persisted to a local YAML state file, enabling topology recovery across restarts. Kafka integration is implemented via the kafka-python library. A TaskProducer service publishes classified inference requests synchronously, invoking flush() to guarantee message persistence prior to returning control to the HTTP handler. Concurrently, a ResultConsumer daemon thread bridges the synchronous Kafka consumer interface with the asynchronous FastAPI event loop through loop.call_soon_threadsafe() primitives.

B. Worker Daemon

The worker daemon is a Python script that runs a registration handshake on startup, then enters a dual-thread loop: one thread polls POST /heartbeat every 5 s; the other runs a Kafka consumer with a 1s poll timeout. On task receipt, the daemon posts to the local Ollama API (http://localhost:11434/api/generate) and publishes the result to the completed tasks Kafka topic. Workers advertise available models by querying Ollama's GET /api/tags at startup and embedding the model list in the registration payload. Worker nodes support dynamic Kafka broker migration through a dedicated signaling mechanism. The heartbeat thread facilitates broker address updates via a thread-safe flag, permitting the primary execution thread to perform consumer and producer swaps between poll cycles. This implementation ensures a seamless transition during cluster topology reconfigurations without interrupting active inference tasks.

C. Frontend

The React application implements six functional views: an authentication suite (Login/Signup) with role-based provisioning, a Mode Selection landing interface, and a Chat component utilizing Server-Sent Events (SSE) for low-latency streaming completions. Administrative oversight is facilitated via a four-tab Dashboard (Overview, Nodes, Users, Requests) featuring a 10-second polling refresh cycle, alongside a Server Status module that renders real-time node telemetry and persistent activity logs. All backend communication is proxied through a single api/index.js module. The module implements a multi-stage leader resolution cascade: a discovery service URL, a localStorage cache of the last-known leader, and a build-time fallback — each health-checked before use. An Axios response interceptor catches 502/503/504 errors, triggers forced leader re-resolution, and transparently retries the failed request, requiring zero user intervention during leader transitions.

D. Docker Compose Deployment

The system orchestration is defined via *docker-compose.yml*, which provisions Kafka in KRaft mode alongside a Cassandra cluster. To ensure high availability, a sidecar initialization container facilitates a persistent retry loop for topic creation, enforcing a replication factor of 1 for single-host development environments to ensure broker availability before proceeding.. This process utilizes *--if-not-exists* flags to maintain idempotent execution across the distributed host environment. Conversely, both the FastAPI leader node and the worker daemons are deployed natively on the host OS rather than within containers; this architectural choice preserves direct access to host CPU instructions and local Ollama socket resources, optimizing inference throughput.

VI. Evaluation

We evaluated the prototype on a four-laptop cluster: Laptop A (leader, 16 GB, Mistral-7B), Laptop B (8 GB, Phi-2), Laptop C (16 GB, Mistral-7B + Deepseek-Coder), and Laptop D (64 GB, Llama-3 70B). All nodes ran macOS 14 or Windows. The Tailscale network operated over a shared Wi-Fi access point with ~15 ms intra-cluster RTT.

A. Baseline Inference Latency

Table II reports median time-to-first-token (TTFT) and total generation time for a 256-token prompt across model tiers. Kafka routing overhead adds a constant ~800 ms that dominates for fast models but is negligible relative to Llama-3 70B generation time.

TABLE II. INFERENCE LATENCY - 256-TOKEN PROMPT

Model	Worker RAM	TTFT (ms)	Total (s)
Phi-2	8 GB	1,240	3.8
Mistral-7B	16 GB	1,890	7.2
Llama-3 70B	64 GB	4,600	41.5

B. Worker Failure Recovery

We simulated worker failure by closing the lid of Laptop C mid-task (Mistral-7B, 512-token prompt). The leader detected the missed heartbeat at T+16 s. Kafka requeued the task at T+34 s. Laptop A completed the task at T+52 s. Total

user-visible overhead: ~45 s, attributable entirely to Kafka's consumer timeout window. Reducing session.timeout.ms to 10 s would lower recovery latency to ~28 s.

C. Leader Failover

We killed the leader process on Laptop A. Workers detected the outage within 5–8 s. Election completed in 3.1 s on average across five trials ($\sigma = 0.8$ s). The Discovery Service updated within 1 s. The React frontend reconnected in 2.4 s. Total service interruption: 10.5 s median. Table III summarises failover timing across all phases.

TABLE III. LEADER FAILOVER TIMING - 5 TRIALS

Phase	Median (s)	Std Dev (s)
Failure detection (workers)	6.2	0.7
Election completion	3.1	0.8
Discovery Service update	0.9	0.2
Frontend reconnect	2.4	0.5
Total service interruption	10.5	1.1

D. Routing Correctness

30 prompts were submitted across three RAM tiers and verified dispatched to eligible workers only. No prompt was incorrectly routed to an under-resourced node. Three prompts requiring Llama-3 70B when Laptop D was offline were correctly queued until Laptop D re-registered, confirming the eligibility filter prevents routing to incapable workers.

VII. Discussion

A. CAP Theorem Positioning

The system prioritises Availability and Partition Tolerance over strong Consistency (AP in the CAP sense [8]). During a network partition, each isolated sub-cluster continues serving requests independently. Upon restoration of connectivity, nodes synchronize membership state through the *POST /cluster/sync* mechanism and subsequently re-converge on the highest-priority active node. For AI inference, idempotent from the user's perspective, this trade-off is acceptable. Applications requiring exactly-once semantics would need full Raft or Paxos.

B. Scalability Considerations

The current leader is a single-process FastAPI application that does not horizontally scale. At high request rates, the Kafka producer and worker registry become bottlenecks. Partitioning the leader role across multiple consumer groups with a frontend is a natural extension. Cassandra's ring architecture already supports horizontal read/write scaling for the data layer.

C. Security

Tailscale's mutual TLS ensures only enrolled devices join the cluster. Apache Cassandra handles user authentication via password hashing and role-based access control, implementing three distinct privilege tiers to enforce secure cluster interaction. The prototype lacks TLS on intra-cluster Kafka traffic and does not encrypt Cassandra data at rest. Production deployments would require mTLS on Kafka brokers and Cassandra client connections. The prototype does not implement stateless token-based authentication (e.g., JWT); production deployments would benefit from this addition.

D. Limitations

Worker availability is non-deterministic, laptop owners may close machines at any time, and the system provides no SLA on task completion. The token-count routing skill heuristic does not model context-length requirements or per-token generation speed. A scheduler incorporating Ollama's reported model load time and per-token latency could optimize for TTFT rather than raw RAM capacity. The convergent election protocol assumes eventually-consistent membership state; under sustained partitions, isolated sub-clusters may serve requests independently until partition healing triggers state merge and re-convergence

VIII. Conclusion

We presented the Distributed AI Gateway, a fault-tolerant cluster for federated LLM inference over a VPN-connected pool of commodity laptops. The system demonstrates that open-weight models can be served reliably from consumer hardware through Kafka-backed task queuing, heartbeat-driven failure detection, and lightweight leader election with cloud based dynamic re-routing. Evaluation on a four-node prototype confirms sub-11-second leader failover, automatic worker task requeue, and correct Ram-Tier routing across three RAM tiers.

The architecture offers a practical, cost-effective, and privacy-preserving alternative to cloud inference APIs for resource-constrained teams. Future work will address horizontal leader scaling, Phi Accrual adaptive heartbeat thresholds, mTLS hardening of intra-cluster communication, and a token-speed-aware scheduling model optimising for time-to-first-token rather than raw RAM capacity.

References

[1] W. Kwon et al., "Efficient Memory Management for Large Language Model Serving with Paged Attention," in Proc. ACM SOSP, 2023.

[2] G. Yu et al., "Orca: A Distributed Serving System for Transformer-Based Generative Models," in Proc. USENIX OSDI, 2022.

[3] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in Proc. USENIX ATC, pp. 305–320, 2014.

[4] N. Hayashibara, X. Défago, R. Yared, and T. Katayama, "The φ Accrual Failure Detector," in Proc. IEEE SRDS, 2004.

[5] Tailscale Inc., "How Tailscale Works," https://tailscale.com/blog/how-tailscale-works, 2020.

[6] Ollama, "Ollama: Run Large Language Models Locally," https://ollama.com, 2023.

[8] E. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed," IEEE Computer, vol. 45, no. 2, pp. 23–29, Feb. 2012.

[7] Apache Kafka, "Kafka Documentation," https://kafka.apache.org/documentation, 2024.

[8] Apache Cassandra, "Cassandra Architecture," https://cassandra.apache.org/doc/latest/architecture, 2024.

[9] Meta AI, "Llama 3.1 Model Card," https://ai.meta.com/blog/meta-llama-3-1/, 2024.