Project Proposal: Distributed LLM Inference Gateway

Li Cao

April 9, 2026

Contents

1 Project Overview

This project implements a Distributed LLM Inference Gateway—a high-performance reverse proxy that routes client inference requests to a cluster of LLM serving replicas. The gateway provides KV-cache-aware routing via consistent hashing, weighted load balancing, fault tolerance with mid-stream failover and request hedging, circuit breaker for degraded replica detection, streaming token delivery, backpressure management, and zero-downtime rolling updates. Replicas participate in a SWIM gossip protocol for decentralized membership and failure detection, rather than relying on a centralized health monitor.

1.1 Motivation

Large Language Model (LLM) inference is one of the most resource-intensive workloads in modern distributed systems. Production LLM serving systems must handle thousands of concurrent requests across multiple GPU-equipped replicas while maintaining low latency and high availability. The infrastructure that sits between clients and model replicas—the inference gateway—is a rich distributed systems problem involving load balancing, fault tolerance, failure detection, flow control, and streaming communication.

1.2 Scope

This project focuses entirely on the distributed systems infrastructure, not on machine learning. LLM backends are simulated: each replica is a lightweight C++ server that receives a prompt, simulates token generation with configurable latency (50–200ms per token), and streams tokens back to the gateway. This design choice allows me to:

- Eliminate dependencies on ML frameworks or GPU hardware.
- Precisely control replica behavior for deterministic testing (e.g., making a replica "slow" to test load balancing, or killing it mid-stream to test failover).
- Focus development effort on the distributed systems challenges.

This project demonstrates the following core distributed systems concepts:

- **Gossip protocol / epidemic dissemination**: SWIM-style protocol for decentralized membership and failure detection among replicas.
- **Fault tolerance**: Replica failure detection via gossip protocol, mid-stream failover during active streaming.
- **Eventual consistency**: Gossip-based membership views converge eventually; the gateway and replicas may temporarily have different views of cluster membership.
- **Consistent hashing**: KV-cache-aware affinity routing using a consistent hash ring for minimal disruption on replica join/leave.
- **Load balancing**: Weighted least-connections routing across heterogeneous replicas.
- **Circuit breaker**: Detecting degraded (not just crashed) replicas via error rate tracking, with automatic recovery probing.
- **Speculative execution**: Request hedging to reduce tail latency by racing two replicas and returning the faster response.
- **Flow control / backpressure**: Request queuing when replicas are saturated.
- **Replication for availability**: Multiple replicas serving the same model.
- **Streaming communication**: gRPC server-side streaming for incremental token delivery.

2 Functional Description

2.1 System Architecture

The system consists of three types of processes, as shown in Figure 1. Replicas communicate with each other via a gossip protocol over UDP, while client-gateway and gateway-replica communication uses gRPC:

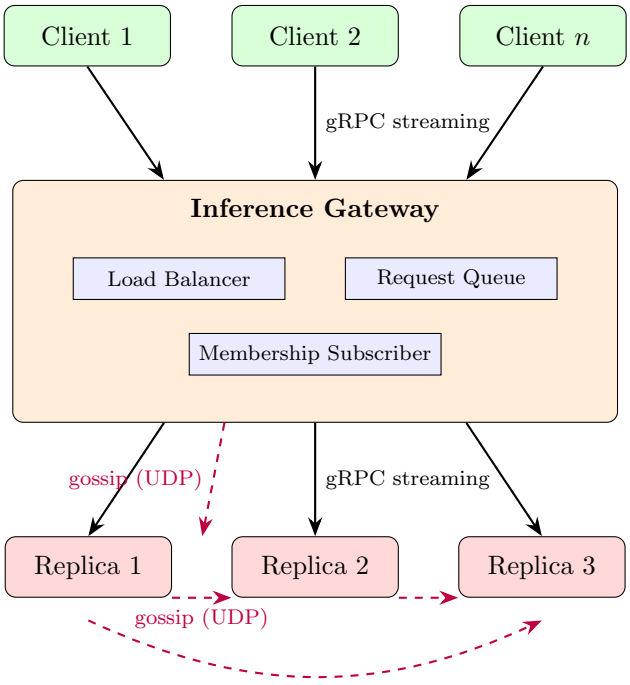

Figure 1: System architecture. Solid arrows show gRPC communication; dashed purple arrows show gossip protocol messages (UDP). Replicas gossip membership state among themselves and to the gateway.

1. **Clients** submit inference requests (prompt text + configuration) to the gateway and receive a stream of generated tokens in response.
2. **Gateway** accepts client connections, selects a backend replica using the load balancer, forwards the request, and proxies the streamed token response back to the client. The gateway subscribes to the gossip network to learn which replicas are alive, rather than actively probing them. It manages request queues and handles mid-stream failover.
3. **Replicas** are independent servers that each simulate an LLM. In addition to serving inference requests, each replica participates in a SWIM-style gossip protocol: replicas periodically ping random peers, disseminate membership updates (joins, failures, suspicions), and report the cluster view to the gateway.

2.2 Interaction Model

The system uses two communication layers:

gRPC (TCP) for client-facing and inference traffic:

- **Client → Gateway**: Server-side streaming RPC. The client sends a single `InferRequest` and receives a stream of `InferResponse` messages, each containing one token.

- **Gateway → Replica**: Server-side streaming RPC. The gateway sends a `GenerateRequest` and receives a stream of `GenerateResponse` messages.
- **Gateway → Replica (Admin)**: Unary RPCs for drain/undrain operations during rolling updates.

UDP for the gossip protocol:

- **Replica ↔ Replica**: Periodic ping/ping-req/ack messages for failure detection (SWIM protocol).
- **Replica → Gateway**: Membership update notifications piggybacked on gossip messages, informing the gateway of joins, failures, and suspicions.

The gateway runs multiple concurrent streaming sessions simultaneously using C++ threading.

3 Design Requirements

I define six design requirements (DR1–DR6) that specify the distributed systems behavior the implementation must satisfy. Each test case (Section 6) maps directly to one or more of these requirements.

3.1 DR1: Request Routing and Load Balancing

The gateway maintains a *replica registry*—a data structure tracking all known replicas, their membership state (as learned from the gossip protocol), capacity weight, and current number of active (in-flight) requests. When a new client request arrives, the gateway selects a replica using a two-tier routing strategy:

Tier 1: KV-cache-aware affinity via consistent hashing. In real LLM serving, replicas cache intermediate key-value tensors from previous requests; routing a follow-up request to the same replica avoids recomputation. The gateway places all ALIVE replicas on a consistent hash ring. When a request arrives, the prompt prefix is hashed to a position on the ring, and the request is routed to the nearest replica (clockwise) that has available capacity. If that replica is at capacity, the gateway continues walking the ring to the next candidate.

Consistent hashing ensures that when a replica joins or leaves the cluster, only a minimal fraction ($\sim 1/N$) of the prompt-to-replica mappings are disrupted, rather than all of them being reshuffled. This stability is critical for maintaining KV cache hit rates during rolling updates and failure recovery.

Tier 2: Weighted least-connections fallback. If the consistent hash ring walk finds no replica with available capacity (e.g., all replicas near the hash position are saturated), the gateway falls back to a weighted least-connections algorithm across all alive replicas:

1. For each ALIVE replica r, compute a score:

$$\text{score}(r) = \frac{\text{active\_requests}(r)}{\text{weight}(r)}$$

2. Select the replica with the lowest score (ties broken randomly).

This fallback ensures that requests are never unnecessarily queued when *some* replica has capacity, even if it is not the ideal affinity target. If no replica has capacity under either tier, the request is queued (DR5).

3.2 DR2: Token Streaming

LLM inference produces tokens incrementally over several seconds. Rather than waiting for the full response, the gateway streams each token to the client as it is generated:

1. The gateway opens a streaming RPC to the selected replica.
2. As the replica generates each token, it sends a `GenerateResponse` to the gateway.
3. The gateway immediately forwards the token as an `InferResponse` to the client.
4. This continues until the replica sends a message with `is_final = true`.

The gateway must support multiple concurrent streaming sessions—many clients may be generating simultaneously, each connected to a different (or the same) replica.

3.3 DR3: Gossip-Based Membership and Failure Detection

Rather than a centralized health monitor polling replicas, failure detection is performed in a decentralized fashion using a SWIM-style gossip protocol among the replicas themselves. This is a core distributed systems protocol that provides:

- **Decentralized failure detection**: No single point of failure for health monitoring. Each replica independently monitors a subset of peers.
- **Epidemic dissemination**: Membership updates (join, suspect, dead) propagate through the cluster via piggybacked gossip messages.
- **Eventual consistency**: All nodes eventually converge to the same membership view, but may temporarily disagree.

The SWIM protocol operates as follows:

1. Each replica maintains a *membership list* of all known replicas with their state: ALIVE, SUSPECT, or DEAD.
2. Every T_{protocol} milliseconds (default: 500ms), a replica selects a random peer and sends a `PING` message via UDP.
3. If the peer responds with an `ACK` within a timeout T_{ping} (default: 200ms), it is confirmed ALIVE.
4. If no `ACK` is received, the replica performs an indirect probe: it selects k random other peers (default: $k = 2$) and asks them to ping the suspect on its behalf via `PING_REQ` messages.
5. If none of the k indirect probes yield an `ACK`, the target is marked SUSPECT.
6. A SUSPECT replica that does not refute the suspicion within T_{suspect} (default: 2 seconds) is declared DEAD and removed from the membership list.
7. Membership state changes are piggybacked on all gossip messages, so they propagate epidemically through the cluster without extra message overhead.

The three membership states and their transitions are illustrated in Figure 2. The key correctness property is the SUSPECT → ALIVE refutation path, which prevents false eviction of slow but alive replicas.

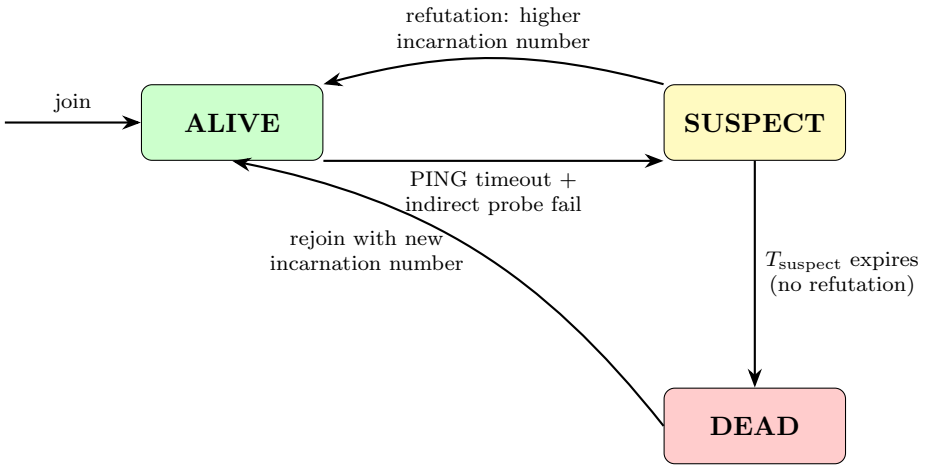

Figure 2: SWIM membership state machine. A falsely suspected replica increments its incarnation number to refute the suspicion, preventing false eviction.

The gateway participates in the gossip network as a non-replica member: it receives piggybacked membership updates from replicas and maintains its own membership view, but does not serve inference requests. When the gateway learns that a replica has transitioned to DEAD, it removes the replica from its routing pool. When a new replica joins (or a previously dead replica recovers), the gateway learns of this through gossip and adds it back.

In addition to membership state, replicas piggyback load metadata (current `active_requests` and `max_capacity`) on gossip messages. The gateway's load balancer uses this gossip-propagated load data, in addition to its own tracking of in-flight requests, to compute more accurate routing scores. This provides a distributed view of replica load without requiring a centralized monitoring service.

The gateway also implements a circuit breaker for each replica. While the gossip protocol detects crashed replicas (process death), the circuit breaker handles *degraded* replicas—those that are still alive but returning errors or responding abnormally slowly. The circuit breaker tracks the error rate and latency of recent requests to each replica. When the error rate exceeds a threshold, the circuit "opens" and the gateway stops routing to that replica. Periodically, the gateway sends a single probe request ("half-open" state) to check if the replica has recovered. If the probe succeeds, the circuit closes and normal routing resumes. This provides a complementary layer of fault detection: gossip detects crashes, while the circuit breaker detects degradation.

3.4 DR4: Mid-Stream Failover

If a replica fails while actively streaming tokens to a client, the gateway must recover:

1. The gateway detects the broken stream (gRPC stream error or timeout).
2. The gateway records how many tokens were already delivered to the client.
3. The gateway selects another ALIVE replica (from its gossip-derived membership view) and sends a new `GenerateRequest` with the `tokens_already_generated` field set, so the new replica can skip ahead.
4. The gateway resumes streaming tokens to the client from the new replica.

The client may observe a brief pause during failover but should not receive an error or experience a hang. The `InferResponse` metadata includes the `replica_id` field so the client can observe that a failover occurred. The gossip protocol will independently detect and disseminate this failure,

ensuring the dead replica is also removed from the membership view of all other replicas.

For latency-sensitive requests, the gateway supports request hedging (speculative execution). Inspired by Google's "The Tail at Scale" approach, the gateway can send the same request to two replicas simultaneously and stream the response from whichever replica produces the first token faster, cancelling the slower one. This reduces tail latency at the cost of extra replica utilization, and is configurable (disabled by default, enabled per-request via a flag in `InferRequest`).

3.5 DR5: Backpressure and Request Queuing

Each replica has a configurable maximum concurrent request limit (e.g., 2 simultaneous streams). When all replicas are at capacity:

1. Incoming requests are placed into a FIFO request queue in the gateway.
2. The queue has a configurable maximum size. Requests that arrive when the queue is full receive an overload error immediately.
3. When a replica completes a request and has capacity, the gateway dequeues the next waiting request and dispatches it.

This prevents replicas from being overwhelmed and provides predictable behavior under load.

3.6 DR6: Rolling Replica Update

The gateway supports zero-downtime rolling updates of replicas (e.g., deploying a new model version):

1. An administrative command initiates a drain on a target replica: the gateway stops sending new requests to the replica but allows in-flight requests to complete.
2. Once all in-flight requests complete, the replica is taken offline.
3. The replica is restarted with a new model version tag and re-joins the gossip network.
4. Other replicas and the gateway learn of the re-join through gossip protocol dissemination.
5. This process is repeated for each replica in sequence.

During the entire rolling update, the remaining replicas continue serving traffic. No requests are dropped. The gossip protocol naturally handles the membership changes (leave and re-join) without requiring any centralized coordination.

4 Approach

4.1 Programming Language

I will implement the project in C++17. C++ is well-suited for this project because:

- The gateway is a high-performance networking component where latency and memory efficiency matter.
- gRPC has mature, first-class C++ support with both synchronous and asynchronous APIs.
- C++ provides fine-grained control over threading and synchronization, which is essential for managing concurrent streaming sessions and the gossip protocol's UDP messaging.
- UDP socket programming for the gossip layer is straightforward in C++ with POSIX sockets.

4.2 Third-Party Libraries

I emphasize that gRPC and Protobuf are used solely for the communication layer. Every component that embodies a distributed systems concept—the gossip protocol, membership management, consistent hash ring, load balancer, circuit breaker, request hedging, request queue, failover logic,

Library	Purpose	Scope of Use
gRPC	RPC framework	Communication transport only
Protocol Buffers	Message serialization	Message encoding/decoding only

Table 1: Third-party libraries.

and drain mechanism—is implemented entirely by me. The gossip protocol uses raw UDP sockets with Protobuf-serialized messages.

4.3 Build System

- CMake for configuring the C++ build, managing dependencies, and compiling protobuf definitions.
- A top-level Makefile with an `all` recipe that builds all binaries and runs the full test suite, as required by the autograder. The Makefile will also handle installing gRPC/Protobuf if not already present.

4.4 Concurrency Model

The system uses standard C++17 concurrency primitives:

- `std::thread` for the gossip protocol thread (periodic pinging), the gateway's membership subscriber thread, and per-client streaming sessions.
- `std::shared_mutex` with `std::shared_lock`/`std::unique_lock` for the membership list and replica registry, which are read-heavy (many concurrent routing lookups) but infrequently written (only on membership changes). This reader-writer lock avoids contention between concurrent streaming sessions.
- `std::mutex` and `std::condition_variable` for the request queue (producer-consumer pattern: streaming completions notify the queue dispatcher).
- `std::atomic<int>` for per-replica active request counters, allowing lock-free updates on the hot path (every request dispatch and completion).

5 Protocol and Service Definitions

5.1 gRPC Services

The system defines two gRPC services for inference and administration:

Listing 1: gRPC service definitions

```
syntax = "proto3";
package llmgateway;

// Client <-> Gateway
service InferenceGateway {
  rpc Infer (InferRequest) returns (stream InferResponse);
}

// Gateway <-> Replica (inference + admin)
service LLMReplica {
  rpc Generate (GenerateRequest) returns (stream GenerateResponse);
```

```
  rpc Drain (DrainRequest) returns (DrainResponse);
}

message InferRequest {
  string client_id = 1;
  string prompt = 2;
  int32 max_tokens = 3;
  bool hedge = 4;           // enable request hedging
}

message InferResponse {
  string token = 1;
  bool is_final = 2;
  string replica_id = 3;
}

message GenerateRequest {
  string request_id = 1;
  string prompt = 2;
  int32 max_tokens = 3;
  int32 tokens_already_generated = 4;
}

message GenerateResponse {
  string token = 1;
  bool is_final = 2;
}

message DrainRequest {}
message DrainResponse { bool success = 1; }
```

5.2 Gossip Protocol Messages

The SWIM gossip protocol uses UDP with Protobuf-serialized messages:

Listing 2: Gossip protocol message definitions

```
syntax = "proto3";
package llmgateway.gossip;

enum MessageType {
  PING = 0;
  PING_REQ = 1;
  ACK = 2;
}

enum MemberState {
  ALIVE = 0;
  SUSPECT = 1;
  DEAD = 2;
}

// Core gossip message sent over UDP
message GossipMessage {
```

```
  MessageType type = 1;
  string sender_id = 2;
  string target_id = 3;       // for PING_REQ: who to probe
  uint64 sequence_num = 4;

  // Piggybacked membership updates (epidemic dissemination)
  repeated MembershipUpdate updates = 5;
}

message MembershipUpdate {
  string member_id = 1;
  string address = 2;         // host:port
  MemberState state = 3;
  uint64 incarnation = 4;     // monotonic counter to resolve conflicts
  string model_version = 5;
  int32 active_requests = 6;  // piggybacked load info
  int32 max_capacity = 7;
}
```

The `incarnation` number is a key part of the SWIM protocol: when a replica receives a SUSPECT message about itself, it increments its incarnation number and broadcasts an ALIVE message. Higher incarnation numbers override lower ones, allowing a falsely suspected replica to refute the suspicion.

6 Testing Plan

I design nine non-trivial tests, each targeting one or more design requirements. Each test involves multiple interacting distributed components and goes well beyond simple connectivity or response checks.

6.1 Test 1: Load Balancing and Consistent Hashing (25 pts) [DR1, DR2]

Setup. Launch 3 replicas with equal capacity weights. Allow gossip to converge. Send 30 concurrent inference requests with diverse prompts, then send 10 requests with identical prompt prefixes.

Procedure. Phase A: 30 requests with unique prompts (5 tokens each at 50ms). Phase B: 10 requests sharing the same prompt prefix. Then kill one replica and resend the same 10 prompts from Phase B.

Verification.

- Phase A: each replica serves 10 ± 2 requests. All 30 requests complete with valid token streams.
- Phase B: requests with the same prompt prefix are routed to the same replica (consistent hashing affinity).
- After killing one replica: requests previously routed to the dead replica are redistributed, but requests whose affinity target is still alive continue going to the same replica ($\sim 2/3$ of mappings preserved).

Why non-trivial. Tests both load balancing fairness under concurrent load and the consistent hashing property that replica removal disrupts only $\sim 1/N$ of prompt-to-replica mappings. Also verifies token streaming (DR2) works correctly across all sessions.

6.2 Test 2: Gossip-Based Failure Detection (30 pts) [DR3]

Setup. Launch 5 replicas and the gateway. Allow gossip to converge so all nodes have a consistent membership view.

Procedure. Kill one replica (terminate its process). Wait for gossip protocol to detect and disseminate the failure.

Verification.

- Within the expected detection time ($T_{\text{protocol}} + T_{\text{ping}} + k \cdot T_{\text{ping}} + T_{\text{suspect}} \approx 3$ seconds), the dead replica is marked DEAD by its peers.
- The gateway receives the membership update through gossip and removes the replica from its routing pool.
- All surviving replicas agree that the dead replica is DEAD (consistent membership view).
- Requests sent after detection are routed only to the 4 surviving replicas.

Why non-trivial. Tests the full SWIM protocol pipeline: direct ping failure, indirect probe (ping-req) through k peers, suspicion dissemination, and eventual declaration of death—all in a decentralized fashion with no central coordinator.

6.3 Test 3: Gossip Protocol Consistency and Convergence (30 pts) [DR3]

Setup. Launch 5 replicas. Allow gossip to converge.

Procedure. Simultaneously kill 2 replicas. Then, after 3 seconds, add a brand-new replica that joins the gossip network.

Verification.

- All 3 surviving original replicas eventually agree on the same membership list: 2 dead, 3 alive (including the new joiner).
- The new replica learns the full membership (including the 2 dead nodes) through piggybacked gossip updates within a bounded convergence time.
- The gateway's membership view converges to match the replicas' view.
- No false positives: the 3 surviving replicas are never falsely marked as suspect or dead.

Why non-trivial. Tests the epidemic dissemination property of gossip under concurrent failures and membership changes. The new replica must bootstrap its membership view from gossip alone. The system must handle the combination of failures and joins without entering an inconsistent state.

6.4 Test 4: Mid-Stream Failover (25 pts) [DR3, DR4]

Setup. Launch 3 replicas. Client sends a request expecting 20 tokens.

Procedure. After the client has received 10 tokens from replica 1, kill replica 1's process.

Verification.

- The gateway detects the broken stream and re-routes to another healthy replica.
- The client receives a complete response of ≥ 20 tokens total without an error.
- The `replica_id` field in the response stream changes partway through, confirming failover occurred.
- The client does not hang or timeout.
- The gossip protocol independently detects and disseminates the failure to all remaining replicas.

Why non-trivial. Exercises failure during an active streaming operation. The gateway must detect the broken stream, preserve the context of partially delivered tokens, and seamlessly resume

generation on a different replica. Meanwhile, the gossip protocol must correctly propagate the failure.

6.5 Test 5: Backpressure Under Saturation (25 pts) [DR5]

Setup. Launch 2 replicas, each configured with a maximum of 2 concurrent requests (total cluster capacity: 4). Configure the gateway queue with a maximum size of 10.

Procedure. Send 8 requests simultaneously. Each request generates 10 tokens at 100ms each ($\sim$1 second per request).

Verification.

- The first 4 requests are immediately dispatched to the replicas (2 per replica).
- The remaining 4 requests are queued in the gateway.
- As each request completes, a queued request is dispatched.
- All 8 requests complete successfully.
- Requests are dequeued and dispatched in FIFO order (verified by comparing request IDs to completion order relative to their batch).

Why non-trivial. Tests the interaction between the request queue, the load balancer, and concurrent request completion events. Requires correct synchronization: the queue must be notified when a replica has capacity, and the dispatch must be atomic with respect to the capacity check.

6.6 Test 6: Suspicion Refutation (Incarnation Numbers) (15 pts) [DR3]

Setup. Launch 3 replicas. Introduce artificial network delay on one replica's gossip port so that it intermittently misses ping deadlines, causing it to be suspected.

Procedure. Observe the gossip protocol's behavior over 10 seconds while the slow replica is still running and responsive (just slow on gossip).

Verification.

- The slow replica is marked SUSPECT by at least one peer.
- The slow replica detects the suspicion (via piggybacked updates) and refutes it by incrementing its incarnation number and broadcasting an ALIVE message.
- After refutation, the replica transitions back to ALIVE in all peers' membership lists.
- The replica is not incorrectly declared DEAD (no false positive).
- The gateway continues routing requests to this replica throughout (with possible brief deprioritization during the SUSPECT window).

Why non-trivial. Tests the incarnation number mechanism, which is the key correctness property of the SWIM protocol. Without it, slow but alive replicas would be incorrectly evicted, causing unnecessary service disruption.

6.7 Test 7: Rolling Replica Update via Gossip (15 pts) [DR3, DR6]

Setup. Launch 3 replicas, all initially reporting model version "v1". Start a background load generator that continuously sends requests throughout the test.

Procedure. Perform a rolling update:

1. Drain replica 1 (gateway stops sending new requests, waits for in-flight to finish).
2. Once drained, shut down replica 1. It leaves the gossip network (detected as dead by peers).
3. Restart replica 1 with model version "v2". It re-joins the gossip network with a new incarnation number.
4. The gateway learns of the re-join through gossip and adds it back to the routing pool.
5. Repeat for replicas 2 and 3.

Verification.

- Zero requests are dropped or fail during the entire rolling update.
- During drain, no new requests are sent to the draining replica.
- After each replica re-joins, gossip disseminates its new version to all peers and the gateway.
- After the update completes, all replicas report version "v2" and all gossip membership views are consistent.
- The background load generator confirms 100% success rate throughout.

Why non-trivial. Tests the interaction between the drain mechanism, gossip-based leave/re-join, and the load balancer under continuous traffic. The gossip protocol must correctly handle the rapid leave-then-rejoin sequence without confusing the old incarnation with the new one.

6.8 Test 8: Circuit Breaker (20 pts) [DR3]

Setup. Launch 3 replicas and the gateway. Configure one replica to return gRPC errors for all `Generate` requests (simulating a degraded but alive replica). The replica remains responsive to gossip pings.

Procedure. Send 20 requests through the gateway over 10 seconds. The degraded replica is alive (gossip sees it as ALIVE) but fails all inference requests.

Verification.

- After the first few failed requests, the circuit breaker opens and the gateway stops routing to the degraded replica.
- Subsequent requests are routed only to the 2 healthy replicas and succeed.
- The degraded replica is not marked DEAD by gossip (it is still responding to pings).
- After fixing the degraded replica (stop returning errors), the circuit breaker transitions to half-open, sends a probe, and resumes routing upon success.

Why non-trivial. Tests the interaction between two independent failure detection layers: gossip (network-level, detects crashes) and circuit breaker (application-level, detects degradation). A system with only gossip would continue routing to the degraded replica indefinitely, causing request failures.

6.9 Test 9: Request Hedging (15 pts) [DR4]

Setup. Launch 2 replicas: one "fast" (50ms per token) and one "slow" (300ms per token). Configure the gateway with hedging enabled.

Procedure. Send 5 hedged requests (with `hedge=true`), each requesting 10 tokens. Each request is sent to both replicas simultaneously.

Verification.

- All 5 responses come from the fast replica (verified via `replica_id`).
- The total time per request is ∼500ms (fast replica's 10×50ms), not ∼3000ms (slow replica's 10×300ms).
- The slow replica's streams are cancelled (verified by checking the slow replica's active request count returns to 0 after each hedged request completes).

Why non-trivial. Tests speculative execution with concurrent streams to two replicas, correct cancellation of the slower stream, and the gateway's ability to race two streaming RPCs and select the winner based on first-token latency.