# Final Report

## Distributed LLM Inference Gateway

Li Cao

April 27, 2026

**Abstract**

Production LLM inference serving requires a gateway layer that load-balances across heterogeneous GPU replicas, streams tokens to clients under second-scale latency budgets, survives mid-stream replica failures, and rolls out new model versions without dropping traffic. Production deployments typically stitch together a routing layer, an external health monitor, and a deployment tool. This report presents the design, implementation, and evaluation of a Distributed LLM Inference Gateway in C++17 that consolidates these functions into a single coherent system.

The design integrates four decentralized-systems techniques that are usually studied in isolation: (i) consistent hashing with virtual nodes for prompt-prefix affinity load balancing, (ii) the SWIM gossip protocol for $O(1)$-per-node membership and failure detection with suspicion and incarnation-based refutation, (iii) speculative request hedging for tail-latency mitigation, and (iv) a circuit breaker layered on top of gossip to absorb gray failures where a replica is alive in gossip but failing inference requests. A ticket-based FIFO backpressure queue and a rolling-update controller that drains replicas one at a time complete the picture, supporting bounded overload behavior and zero-downtime model upgrades.

A nine-test integration suite exercises every component end-to-end through an in-process `TestCluster` harness. All tests pass deterministically across ten full-suite runs. Fourteen SWIM unit tests and four gossip integration tests add complementary test coverage of the membership layer.

# Contents

**1 Introduction** **4**
1.1 Problem Statement . . . . . 4
1.2 Core Distributed Systems Concepts Exercised . . . . . 4
1.3 Report Organization . . . . . 5

**2 Design Requirements and Test Suite** **5**
2.1 Design Requirements . . . . . 5
2.2 Test Suite and Point Allocation . . . . . 5

**3 System Architecture** **6**
3.1 Process Model . . . . . 6
3.2 Communication Planes . . . . . 7
3.3 Port-Based Address Derivation . . . . . 7
3.4 End-to-End Request Walkthrough . . . . . 8

**4 Design Requirement Implementation** **8**
4.1 DR1: Request Routing and Load Balancing . . . . . 8
4.2 DR2: Token Streaming . . . . . 10
4.3 DR3: SWIM Gossip Protocol . . . . . 10
4.3.1 Protocol Overview . . . . . 11
4.3.2 Piggyback Dissemination . . . . . 11
4.3.3 Conflict Resolution Rules . . . . . 12
4.3.4 Self-Refutation . . . . . 12
4.3.5 Gateway as Gossip Subscriber . . . . . 12
4.3.6 Load Metadata Piggyback . . . . . 12
4.4 DR4: Mid-Stream Failover and Request Hedging . . . . . 13
4.4.1 Mid-Stream Failover Retry Loop . . . . . 13
4.4.2 Request Hedging (Speculative Execution) . . . . . 14
4.5 DR5: Backpressure and Request Queuing . . . . . 15
4.6 DR6: Rolling Replica Update . . . . . 16

**5 Test Design and Results** **17**
5.1 Testing Philosophy . . . . . 17
5.2 The TestCluster Harness . . . . . 17
5.3 Test Driver and Output Format . . . . . 18
5.4 Per-Test Specifications . . . . . 18
5.4.1 Test 1: Load Balancing and Consistent Hashing . . . . . 18
5.4.2 Test 2: Gossip Failure Detection . . . . . 19
5.4.3 Test 3: Gossip Convergence . . . . . 19
5.4.4 Test 4: Mid-Stream Failover . . . . . 20
5.4.5 Test 5: Backpressure Under Saturation . . . . . 20
5.4.6 Test 6: Suspicion Refutation . . . . . 21
5.4.7 Test 7: Rolling Update via Gossip . . . . . 21
5.4.8 Test 8: Circuit Breaker . . . . . 22
5.4.9 Test 9: Request Hedging . . . . . 23
5.5 Aggregate Results . . . . . 23
5.6 Stability . . . . . 23

5.7 Additional Test Coverage . . . 24

**6 Concurrency Model and Thread Safety** **24**
6.1 Thread Inventory . . . 24
6.2 Synchronization Primitives . . . 25
6.3 Lock Ordering and Deadlock Avoidance . . . 25
6.4 Cancellation and Lifetime . . . 26

**7 Code Quality and Design Decisions** **26**

**8 Repository Structure and Build** **28**
8.1 File Tree . . . 28
8.2 Build System . . . 29
8.3 Targets Built . . . 29
8.4 `make all` Output . . . 29

**9 Documentation Artifacts** **29**

**10 Mapping to Distributed Systems Concepts** **30**

**11 Known Limitations and Future Work** **30**

**A Full Test Driver Output** **32**

**B Build Requirements and Environment** **32**

# 1 Introduction

## 1.1 Problem Statement

Large Language Model (LLM) inference is among the most resource-intensive workloads in modern distributed systems. Production serving infrastructures route thousands of concurrent prompt requests to heterogeneous GPU-equipped replicas while maintaining low first-token latency and high availability. The piece of infrastructure that sits between clients and model replicas — the inference gateway — is a rich distributed systems problem in its own right. Its responsibilities include:

- Load balancing across replicas with heterogeneous capacity.
- Detecting replica failures and rerouting traffic accordingly.
- Maintaining a consistent view of cluster membership across replicas and the gateway itself.
- Continuing an in-flight streaming session on another replica when the original replica fails mid-stream.
- Absorbing bursts that exceed instantaneous cluster capacity without crashing or dropping requests.
- Orchestrating rolling model upgrades without downtime.

This project focuses entirely on the distributed-systems infrastructure, not on the ML workload itself. LLM backends are simulated: each "replica" is a lightweight C++ service that exposes a gRPC streaming endpoint and emits synthetic tokens at a configurable per-token delay. This separation keeps the focus on the distributed-systems problems while still exercising realistic streaming semantics, timeouts, and cancellation.

## 1.2 Core Distributed Systems Concepts Exercised

The implementation demonstrates and evaluates the following concepts, each of which is tied to specific components and tests later in the report:

- **Gossip protocol.** A SWIM-style protocol among replicas and the gateway, with piggybacked updates, incarnation-number conflict resolution, and suspicion–refutation.
- **Eventual consistency.** Membership views converge over time; different nodes may hold transiently different views.
- **Fault tolerance.** Crash detection via gossip, mid-stream failover at the gateway, circuit breaker for application-level failures, graceful drain for planned departures.
- **Consistent hashing.** Prompt-prefix affinity routing with virtual nodes on a hash ring; minimizes remap churn on replica join/leave.
- **Weighted least-connections.** Capacity-aware fallback when the hash ring is exhausted.
- **Circuit breaker pattern.** Three-state (CLOSED, OPEN, HALF_OPEN) per-replica health guard with sliding-window error tracking and automatic recovery probing.
- **Speculative execution.** Request hedging: issue the same request to two replicas and keep the first responder.
- **Backpressure.** Ticket-based FIFO queue with bounded size.
- **Replication for availability.** Multiple equivalent replicas, any of which can serve any request.
- **Streaming communication.** gRPC server-side streaming with thread-per-stream concurrency.

### 1.3 Report Organization

Section 2 lists the six design requirements and the nine tests that verify them. Section 3 presents the system architecture. Section 4 is the implementation deep-dive, walking through each DR with data structures, algorithms, and code pointers. Section 5 covers the test suite: philosophy, the in-process harness, per-test specifications, and aggregate results. Section 6 describes the concurrency model and thread-safety discipline. Section 7 collects design decisions and trade-offs. Section 8 documents the repository structure, build system, and test invocation. Section 9 lists the generated documentation artifacts. Section 10 maps implementation choices back to distributed systems concepts. Section 11 enumerates known limitations. Section A contains the verbatim test output.

## 2 Design Requirements and Test Suite

### 2.1 Design Requirements

| ID | Name | Summary |
|---|---|---|
| DR1 | Load Balancing | Two-tier routing: consistent hashing for prompt-prefix affinity, weighted least-connections fallback. |
| DR2 | Token Streaming | gRPC server-side streaming; gateway proxies tokens replica → client. |
| DR3 | Gossip Membership | SWIM protocol: PING / PING_REQ / ACK, suspicion–refutation via incarnation numbers, piggybacked dissemination, gateway participates as non-replica member. |
| DR4 | Mid-Stream Failover | Broken stream → select another replica → resume with `tokens_already_generated`. Request hedging races two replicas and keeps the winner. |
| DR5 | Backpressure | Bounded FIFO queue at the gateway when all replicas are at capacity; overload requests receive immediate error. |
| DR6 | Rolling Update | Drain → stop → restart with new version → gossip re-join; no dropped requests. |

**Table 1:** Design requirements overview.

### 2.2 Test Suite and Point Allocation

| # | Test | DRs | Category | Points |
|---|---|---|---|---|
| 1 | Load Balancing + Consistent Hashing | DR1, DR2 | Routing | 25 |
| 2 | Gossip Failure Detection | DR3 | Membership | 30 |
| 3 | Gossip Convergence | DR3 | Membership | 30 |
| 4 | Mid-Stream Failover | DR3, DR4 | Availability | 25 |
| 5 | Backpressure Under Saturation | DR5 | Backpressure | 25 |
| 6 | Suspicion Refutation | DR3 | Membership | 15 |
| 7 | Rolling Update via Gossip | DR3, DR6 | Deployment | 15 |
| 8 | Circuit Breaker | DR3 | Fault tol. | 20 |
| 9 | Request Hedging | DR4 | Latency | 15 |
| | **Total** | | | **200** |

**Table 2:** Test suite mapping to design requirements.

# 3 System Architecture

## 3.1 Process Model

The system consists of three process types: one gateway, $N$ replicas, and $M$ clients. All are implemented in C++17, using gRPC for application-level RPC and raw UDP for gossip. Figure 1 shows the full topology.

**Communication.** Two transport planes carry different traffic. gRPC over TCP carries the streaming `Infer` path from client to gateway and the streaming `Generate` path from gateway to replica. SWIM gossip over UDP runs as a full mesh among the replicas; the gateway also participates in the gossip network as a non-replica member, with the Membership Subscriber component terminating gossip links on the gateway side.

**Gateway-internal components.** Six cooperating components realize the gateway's responsibilities. On the request hot path: the Load Balancer routes each request using two-tier selection (consistent hashing with virtual nodes for prompt-prefix affinity, then weighted least-connections as a fallback); the Circuit Breakers provide per-replica three-state application-level health guarding, so a replica that is alive in gossip but failing inference is temporarily removed from the routing pool; and the Request Queue is a ticket-based strict-FIFO backpressure queue that bounds outstanding requests when the cluster is saturated. For cluster state and orchestration: the Replica Registry is a thread-safe cache of membership and load metadata, and the Rolling Updater drives the drain → stop → restart → rejoin sequence for zero-downtime updates. The Membership Subscriber is the gateway's SWIM participant.

**Internal cooperation.** The components cooperate as follows. The Membership Subscriber writes state-change callbacks into the Replica Registry whenever gossip observes a new ALIVE / SUSPECT / DEAD transition, and the Rolling Updater also writes the Registry to set or clear each replica's draining flag during a rolling update. The Load Balancer reads the Registry on every `SelectReplica` call, and the Rolling Updater additionally triggers a `RebuildRing` on the Load Balancer after every drain and restart to keep the consistent-hash ring in sync with the latest routable set.

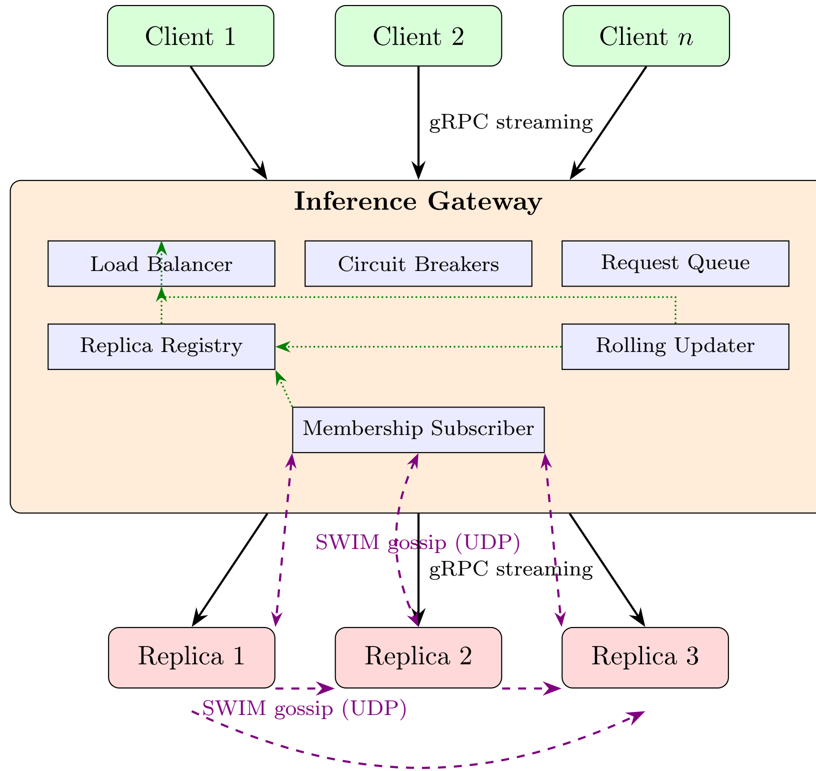


**Figure 1:** System architecture. Solid arrows: gRPC (TCP). Dashed violet: SWIM gossip (UDP). Dotted green: internal data flow inside the gateway. See §3 for the full description of components and arrows.

### 3.2 Communication Planes

Two transport layers carry different kinds of traffic:

- **gRPC over TCP.** Used for client ↔ gateway (streaming `Infer` calls) and gateway ↔ replica (streaming `Generate` calls, plus unary `Drain` for rolling updates). gRPC is chosen over raw TCP because it provides server-side streaming, flow control, cancellation propagation, and deadline handling out of the box.
- **Raw UDP, Protobuf-serialized.** Used for SWIM gossip: PING, PING_REQ, ACK, each of which can piggyback membership updates. UDP fits gossip's traffic pattern. SWIM tolerates packet loss by design — a lost message is recovered on the next round — while TCP's per-connection overhead would dominate the cost of the small, frequent datagrams the protocol exchanges.

### 3.3 Port-Based Address Derivation

Each replica's gRPC port and gossip port are tied by a fixed offset: `grpc_port` = `gossip_port` −10000. Replicas advertise only their gossip endpoint in gossip messages, and the gateway derives the corresponding gRPC endpoint deterministically. This sidesteps a bootstrapping problem — the gateway would otherwise need a separate discovery channel to learn each replica's gRPC address — without introducing a naming server.

### 3.4 End-to-End Request Walkthrough

Figure 2 traces a single inference request through the system on the normal, failure-free path.

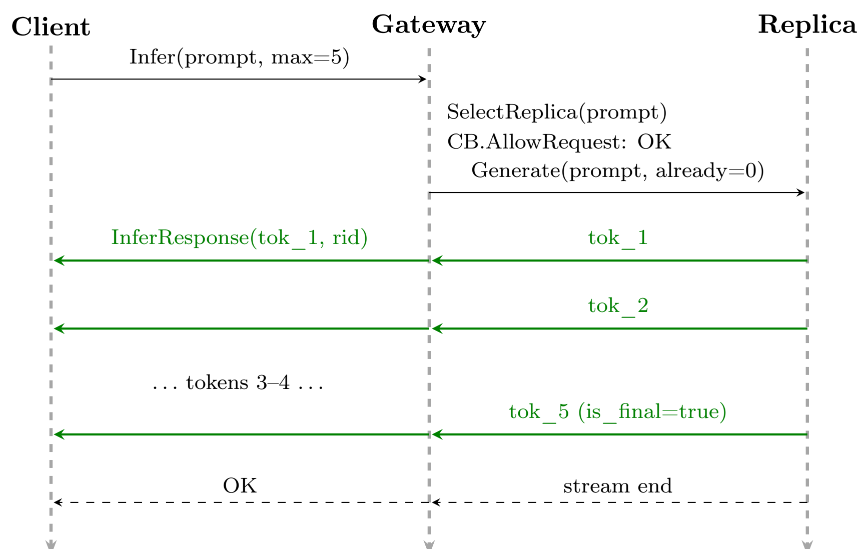


**Figure 2:** Normal-path sequence for a 5-token inference request. The gateway behaves as a streaming reverse proxy: it forwards tokens from replica to client in real time, annotating each response with the serving `replica_id`. Mid-stream failover replaces the replica-side of this diagram with two replicas in succession (section 4.4).

## 4 Design Requirement Implementation

This section walks through how each of the six design requirements is realized in code. For each DR, it identifies the governing component, describes the algorithm or protocol, discusses data structures and invariants, and cites the exact source files where the logic lives.

### 4.1 DR1: Request Routing and Load Balancing

**Goal.** Distribute client requests across the replica pool such that (a) prompts sharing a prefix reliably hit the same replica, and (b) no single replica becomes a hotspot when the hash distribution is skewed.

**Implementation.** The `LoadBalancer` class (`src/gateway/load_balancer.h`, `.cpp`) implements a two-tier strategy.

**Tier 1 — Consistent Hashing with Virtual Nodes.** Each physical replica is placed at 150 virtual positions on a `std::map<size_t, string>` hash ring, keyed by `hash(replica_id + "#" + i)` for $i \in [0, 150)$. Virtual nodes are the standard trick to spread the key space evenly despite only having a handful of physical replicas: with only a few physical keys on the ring, one bad hash could leave a single replica owning a disproportionate arc. 150 virtual nodes per replica drives the relative load imbalance below 10% for clusters of 3–5 replicas, which is tighter than the $\pm 3$ tolerance Test 1 requires.

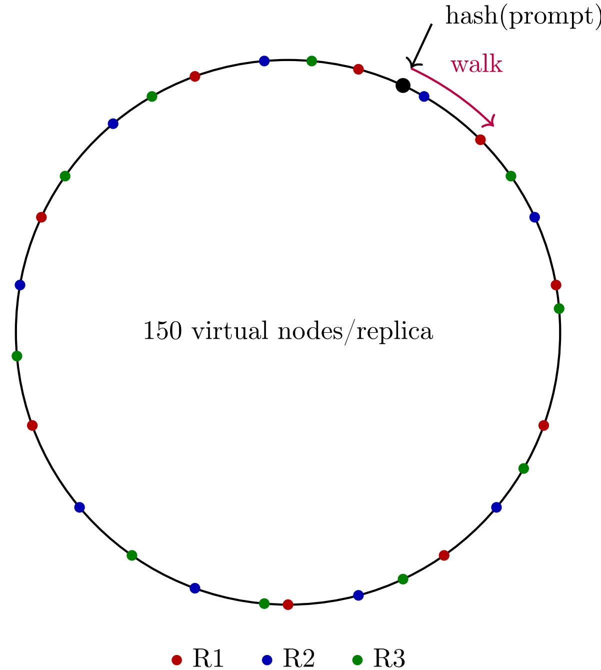


**Figure 3:** Consistent hash ring with virtual nodes. A prompt is hashed to a point on the ring; the selector walks clockwise until it finds a replica with remaining capacity. The density of each replica's virtual nodes evens out the expected load.

`SelectByConsistentHash` walks the ring clockwise from `hash(prompt[:64])` until it finds a replica that satisfies all of: (i) ALIVE-or-SUSPECT in the registry, (ii) not draining, (iii) has remaining capacity (`active_requests` $<$ `max_capacity`, or `max_capacity` $= 0$ meaning unlimited), (iv) not in the caller-supplied exclusion set. If the full loop completes without finding a candidate, it falls through to Tier 2.

**Tier 2 — Weighted Least-Connections.** `SelectByLeastConnections` computes a score for each non-excluded, alive, non-draining replica with remaining capacity:

$$\text{score}(r) = \frac{\texttt{active_requests}(r)}{\text{weight}(r)}$$

and selects the minimum-score replica. Ties (common at startup when every replica has zero active requests) are broken by reservoir sampling so that among $k$ tied candidates each is chosen with probability $1/k$.

**The exclude set.** `SelectReplica` accepts an `std::unordered_set<string> excluded` argument. Consistent-hashing selection is deterministic for a given prompt, so without an exclude set a retry loop would re-select the same target on every iteration: if the first hit has an OPEN circuit or its stream fails, the retry would exhaust `max_retries` on the same broken replica. The gateway populates `excluded` as it learns each replica is unusable for the current request (OPEN circuit, failed stream, partial delivery) and passes it on every subsequent retry. Section 4.4 shows the integration in the retry loop.

**Ring rebuild on membership change.** The ring is not rebuilt on every request. Instead, the gossip subscriber callback triggers `RebuildRing` whenever a membership *state* changes (ALIVE ↔

SUSPECT ↔ DEAD); routine load-metadata updates do not rebuild, because that would block the gossip receiver thread with $150 \times N$ hash computations several times per second.

**Complexity.** `SelectReplica` is $O(\log V)$ for the initial `lower_bound`, then $O(k)$ walk in the amortized-rare case where the first hit is unusable, where $V = 150N$ is the total ring population. `RebuildRing` is $O(V \log V)$.

## 4.2 DR2: Token Streaming

**Goal.** Deliver tokens to the client as they are produced by the replica, not in a batch. Multiple concurrent streams must not block one another.

**Implementation.** The gateway uses gRPC's synchronous server-side streaming API for both ends of its proxy role. The `InferenceGateway` service declares:

```
service InferenceGateway {
  rpc Infer (InferRequest) returns (stream InferResponse);
}
```

The gRPC synchronous API pairs each client stream with one server thread from a thread pool. Inside that thread, `GatewayServer::StreamFromReplica` (`src/gateway/gateway_server.cpp`) does the proxying:

1. Opens a `grpc::ClientReader<GenerateResponse>` to the selected replica.
2. In a loop, calls `reader->Read(&resp)` to pull the next token from the replica.
3. For each token received, constructs an `InferResponse` with the same `token` payload plus the `replica_id` (so the client observes which replica served each token, which is the key observable for failover tests) and calls `writer->Write(...)` to send it to the client.
4. When the replica's stream ends (`reader->Read` returns false), call `reader->Finish()` to capture the gRPC status. A successful status means the replica reported `is_final=true`; an error status triggers failover (section 4.4).

Thread-per-stream is simpler than async gRPC (no completion queues, no state machines) and scales easily to the dozens of concurrent streams a course-project gateway needs to handle. The cost is one OS thread per in-flight request, which is why the gateway also maintains a bounded request queue (section 4.5) to cap the maximum outstanding count.

**Replica-side streaming** is mirrored in `ReplicaServer::Generate`. The replica loops from `tokens_already_generated` to `max_tokens`, sleeps `token_delay_ms` to simulate compute, checks `context->IsCancelled()` (so hedging cancellation takes effect promptly), and writes each token with `is_final` set on the last one. This cancellation-polling loop is what makes mid-stream failover work cleanly: when the gateway kills a slow stream or a hedge loser, the replica exits the loop instead of continuing to compute wasted work.

## 4.3 DR3: SWIM Gossip Protocol

SWIM (Scalable Weakly-consistent Infection-style process group Membership protocol; Das, Gupta, Motivala, DSN 2002) is the core distributed-systems mechanism of this project and is covered in depth here because it has the most moving parts: a two-thread protocol engine, incarnation-based conflict resolution, piggyback dissemination, and self-refutation.

### 4.3.1 Protocol Overview

Every member (replicas and the gateway) maintains a local *membership list* mapping member-id → `{address, state, incarnation, load-metadata}`. States form a three-valued type `ALIVE`, `SUSPECT`, `DEAD`. Incarnation numbers are monotonically increasing per-member integers that break ties during conflict resolution.

Each member runs two threads:

- **Sender thread.** Every `protocol_period_ms` (200ms in the TestCluster configuration) picks a random alive peer, sends it a direct PING, and waits up to `ping_timeout_ms` (100ms) for an ACK. If no ACK arrives, it asks $k$ other random peers (via PING_REQ) to indirectly probe the target; if none of those yield an ACK either, it marks the target `SUSPECT`.
- **Receiver thread.** Listens on the UDP socket. For every incoming message it first processes any piggybacked membership updates, then dispatches on message type (PING → reply with ACK; PING_REQ → forward PING to the target and relay its ACK back; ACK → signal the sender thread).

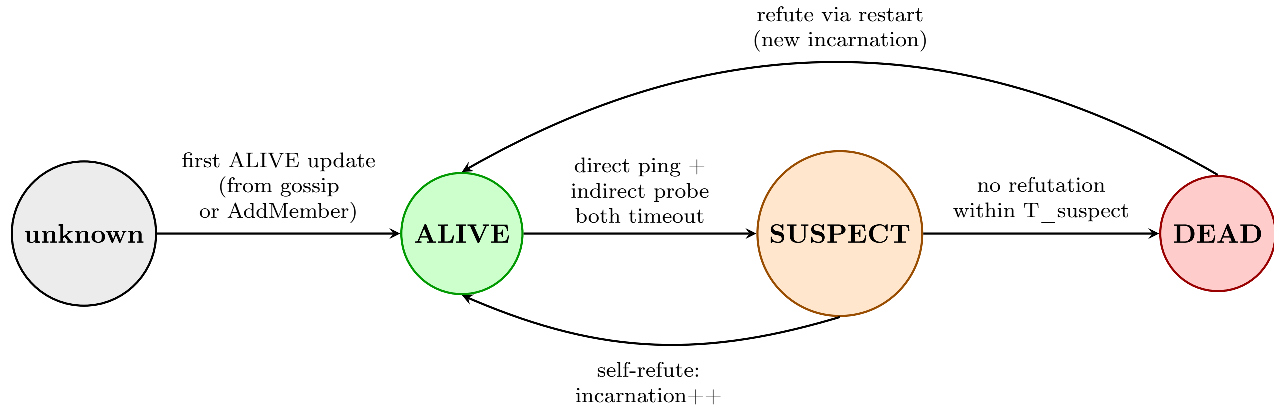


**Figure 4:** SWIM membership state transitions. New members enter as ALIVE. ALIVE → SUSPECT happens when both direct and indirect pings time out. SUSPECT → DEAD happens when `T_suspect` expires with no refutation. The SUSPECT → ALIVE (and DEAD → ALIVE) transition is the refutation path: the member, upon seeing a SUSPECT or DEAD update about itself, increments its incarnation and broadcasts a fresh ALIVE update, which wins under the conflict-resolution rules of section 4.3.3.

### 4.3.2 Piggyback Dissemination

Every outgoing message carries up to `max_piggyback_updates` (default 8) recent membership updates pulled from a local buffer. The sender also attaches its own current state (self-update) as the last piggyback slot. This "free" information channel is how SWIM disseminates state changes — no dedicated gossip rounds are needed. Each update is retransmitted at most `kMaxPiggybackSends` (10) times before eviction from the buffer, giving a bounded dissemination window.

The eviction bound is the usual SWIM trade-off: too small and updates do not reach every peer; too large and the buffer stagnates with stale information. Ten retransmissions over ∼2 seconds is enough for full cluster dissemination in practice with small clusters, at the cost that a member joining after the 2-second window may not learn about the long-gone peer — see the Test 3 discussion in section 5.4.3.

#### 4.3.3 Conflict Resolution Rules

When two conflicting pieces of information about the same member arrive (e.g., peer A says "r3 is ALIVE at incarnation 0" and peer B says "r3 is DEAD at incarnation 0"), the receiver must decide which to keep. The `MembershipList::ApplyUpdate` rules (`src/gossip/membership_list.cpp`) are:

1. **Rule 1 (higher incarnation wins).** If the incoming update's incarnation is strictly greater than the stored incarnation, the incoming update wins regardless of state.
2. **Rule 2 (equal incarnation, stronger state wins).** At the same incarnation, `DEAD` beats `SUSPECT` beats `ALIVE`. This is how a same-round DEAD declaration overrides a stale ALIVE.
3. **Rule 3 (strictly older incarnation is ignored).** Even if the incoming update claims DEAD, a strictly older incarnation is treated as stale and discarded.

Rule 3 is essential. Without it, a DEAD update from an uninformed peer's piggyback buffer could resurrect the dead-then-refuted state of a member that has already bumped its incarnation and returned to ALIVE. Test 3 exercises this path by design: a fresh joiner r6 learns about other members through gossip and could otherwise see transient-SUSPECT piggybacks for r2 / r3 at a stale incarnation. The unit test `test_dead_incarnation_rules` in `tests/test_gossip_unit.cpp` pins this behavior in place with explicit stale-DEAD, equal-DEAD, and higher-ALIVE cases.

#### 4.3.4 Self-Refutation

When a member processes an incoming update about *itself*, it runs `CheckSelfRefutation` before applying the update. If the incoming state is SUSPECT or DEAD, and the incoming incarnation is at least the member's own current incarnation, the member calls `IncrementMyIncarnation`, which bumps the local incarnation and queues a fresh ALIVE self-update for piggybacking. On the next gossip round, that refuted ALIVE update propagates to peers, where Rule 1 causes it to override the SUSPECT/DEAD at the lower incarnation.

This is the piece that protects slow-but-alive members from being incorrectly evicted, and it is the focus of Test 6 (section 5.4.6).

#### 4.3.5 Gateway as Gossip Subscriber

The gateway participates in gossip as a non-replica member. It runs its own `SwimProtocol` instance and registers a membership-change callback that updates the `ReplicaRegistry` and (on state change only) rebuilds the consistent-hash ring. The registry filters its view so that "gateway" and "seed-*" identities are excluded from routing decisions — the gateway never tries to route inference to itself or to a bootstrap placeholder.

#### 4.3.6 Load Metadata Piggyback

In addition to membership state, each replica advertises its current active-request count and max capacity to the rest of the cluster via gossip. Each replica calls `SetMyLoad` periodically with its current values; the next outgoing gossip message carries them in the self-update piggyback. The gateway's registry stores `reported_active_requests` and `max_capacity` and uses them as routing inputs. The membership callback fires not only on state transitions but also when load metadata changes meaningfully, so the registry stays fresh despite the default "no callback on same-state update" SWIM behavior.

## 4.4 DR4: Mid-Stream Failover and Request Hedging

**Goal.** A replica crash or stream error during active token generation must not bubble up to the client as a failure; the gateway must transparently resume the stream on another replica. Additionally, hedging reduces tail latency by racing two replicas and keeping the first responder.

### 4.4.1 Mid-Stream Failover Retry Loop

The `GatewayServer::Infer` handler (`src/gateway/gateway_server.cpp`) runs up to `max_retries` (default 3) passes over the same request. Figure 5 diagrams the fault case.

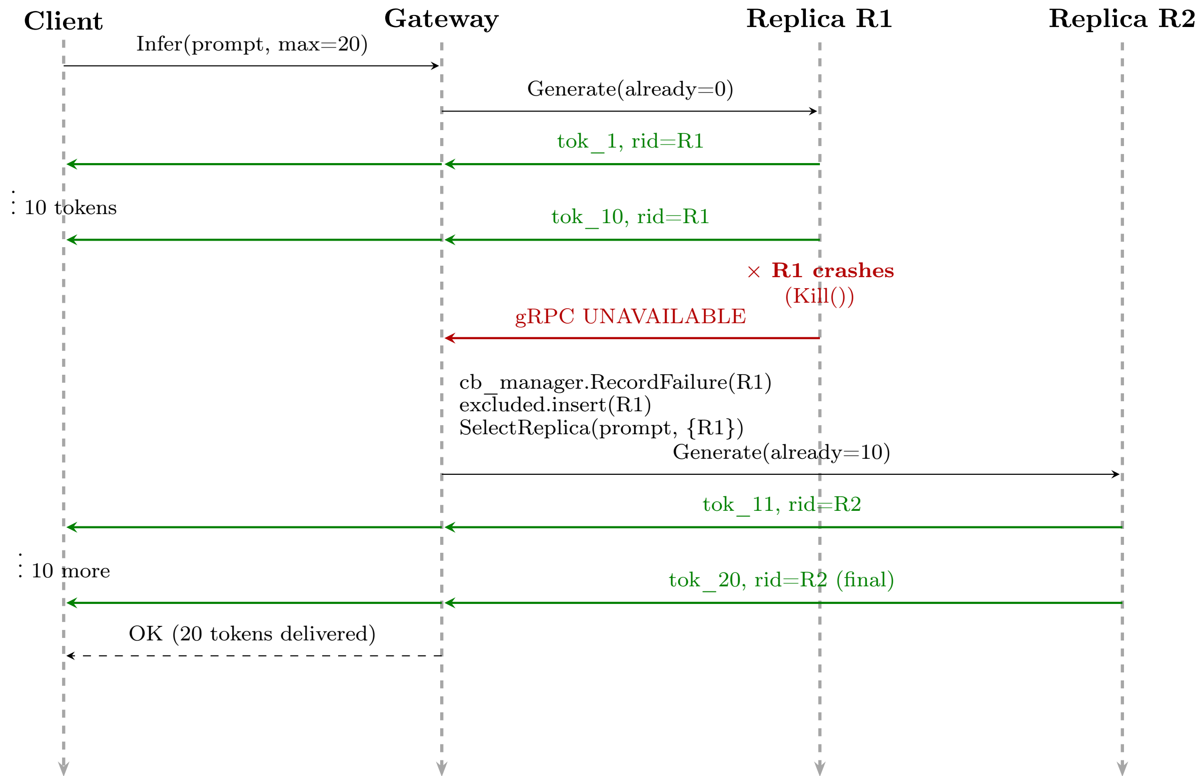


**Figure 5:** Mid-stream failover sequence. R1 serves the first 10 tokens, then crashes. The gateway's `StreamFromReplica` returns −1; the `Infer` retry loop records a circuit-breaker failure, adds R1 to the per-request exclude set, and selects R2 via the load balancer. R2 is told `tokens_already_generated=10` and resumes from token 11. The client sees a single stream of 20 tokens; the only observable change is the `replica_id` field flipping mid-stream.

The retry loop's bookkeeping is:

```
int tokens_sent = 0;
std::unordered_set<std::string> excluded;

for (int attempt = 0; attempt < max_retries && tokens_sent < max_tokens;
     attempt++) {
    auto selection = lb_.SelectReplica(prompt, excluded);
    if (!selection) {
        if (!queue_.WaitForCapacity()) {
            return grpc::Status(grpc::StatusCode::RESOURCE_EXHAUSTED,
                                "no replica with capacity");
```

```
        }
        selection = lb_.SelectReplica(prompt, excluded);
        if (!selection) continue;
    }
    if (!cb_manager_.AllowRequest(selection->replica_id)) {
        excluded.insert(selection->replica_id);
        continue;
    }
    registry_.IncrementActive(selection->replica_id);
    int streamed = StreamFromReplica(
        selection->replica_id, selection->grpc_address,
        prompt, max_tokens, tokens_sent, writer, context);
    registry_.DecrementActive(selection->replica_id);
    queue_.NotifyCapacityAvailable();

    if (streamed < 0) {
        cb_manager_.RecordFailure(selection->replica_id);
        excluded.insert(selection->replica_id);
        continue;
    }
    tokens_sent += streamed;
    cb_manager_.RecordSuccess(selection->replica_id);
    if (tokens_sent >= max_tokens) return grpc::Status::OK;
    excluded.insert(selection->replica_id);  // avoid reselecting after partial
}
```

Key points:

- The `excluded` set is essential: consistent-hashing selection is deterministic for a given prompt, so without exclusions the retry loop keeps picking the same dead replica every iteration and exhausts `max_retries` without making progress.
- `tokens_sent` is threaded through to the replica as `tokens_already_generated`, so the replica resumes the stream at the correct offset rather than restarting from token 0.
- If the loop exhausts retries after partial delivery (some tokens sent but not all), the handler returns `OK` anyway; the client can detect incompleteness by the absence of `is_final=true`. Returning an error status after data has already been written to the wire would be misleading.

### 4.4.2 Request Hedging (Speculative Execution)

When the client sets `InferRequest.hedge=true` and at least two replicas are available, the gateway dispatches to `InferHedged` instead of the normal retry loop. This is speculative execution in the sense of Dean and Barroso's "The Tail at Scale": fire the request to two replicas in parallel, use whichever returns the first token faster, cancel the loser.

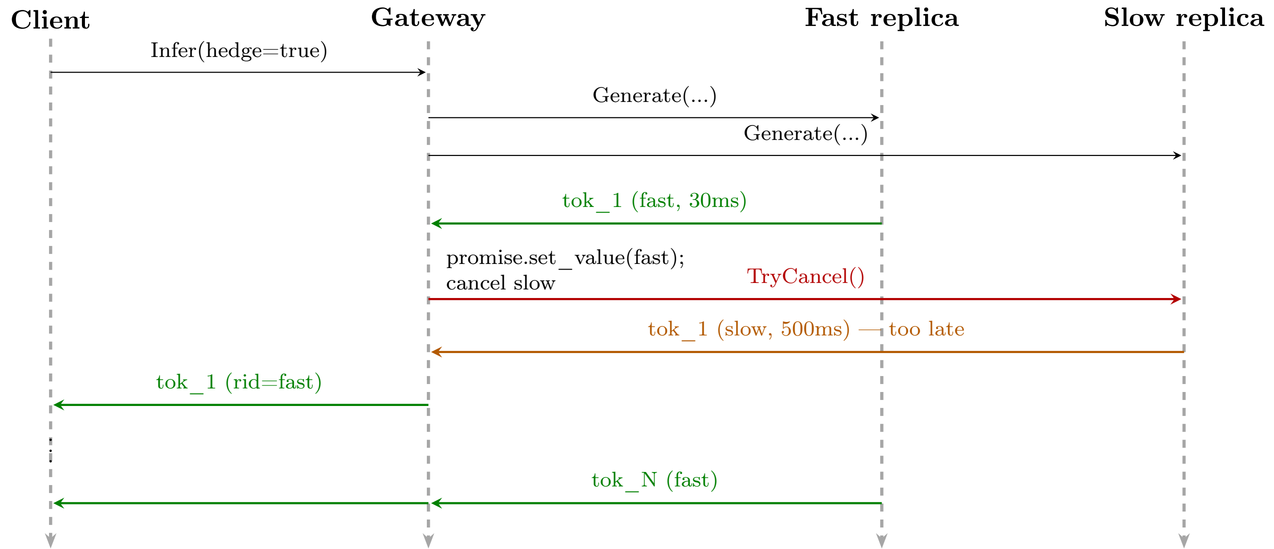


**Figure 6:** Request hedging. Both replicas begin generating. The fast replica produces the first token before the slow one; the gateway publishes the winner via a `std::promise`, cancels the loser's stream with `context.TryCancel()`, and forwards the rest of the winner's tokens to the client normally. The slow replica's Generate loop sees the cancelled context on its next poll and exits, preventing wasted compute.

`InferHedged` selects two different replicas: the first via a normal load-balancer call, the second by hashing a perturbed prompt (`prompt + "_hedge"`) while temporarily inflating the first replica's active-request count so the load balancer routes elsewhere. If the two selections still collide, the gateway falls back to a single-replica stream. The hedge then launches a detached thread for each replica to open a `ClientReader` and pull tokens, and uses a `std::atomic<bool>` compare-exchange to pick the first thread to read a token. The winner's remaining tokens are forwarded to the client; the loser's `grpc::ClientContext` is `TryCancel`-ed so its stream terminates promptly and its resources are released. Test 9 (section 5.4.9) validates the winner selection, the latency savings, and that the loser's `active_requests` counter drains back to zero after cancellation.

## 4.5 DR5: Backpressure and Request Queuing

**Goal.** When all replicas are at their per-replica capacity limit, new requests must neither crash the gateway nor be silently accepted into an unbounded in-memory queue. Instead, the gateway must block the client thread until capacity frees up, subject to a bounded queue size and timeout.

**Implementation.** `RequestQueue` (`src/gateway/request_queue.h`, `.cpp`) uses a ticket-based strict-FIFO condition-variable protocol:

- Each waiting thread atomically takes a monotonic ticket number on entry to `WaitForCapacity`.
- A single `serving_ticket_` counter increments on every `NotifyCapacityAvailable` call (fired by the gateway when a replica finishes a request).
- The `condition_variable::wait_until` predicate is "`serving_ticket_` > my ticket". When the notifier broadcasts, threads re-evaluate their predicates and only the one with the matching ticket proceeds. This gives strict FIFO ordering even on implementations where `notify_one` does not inherently wake the longest-waiting thread.

When the queue is full (more than `max_size` waiters) or the 30-second timeout elapses,

WaitForCapacity returns false and the gateway responds to the client with RESOURCE_EXHAUSTED — the standard gRPC overload signal that lets the client back off or queue locally.

**Integration with the dispatch path.** The gateway's Infer handler calls lb_.SelectReplica(prompt, excluded); if no replica has capacity the selector returns nullopt, and the handler then calls queue_.WaitForCapacity(). On wake it retries selection. After any successful stream completes (including partial), queue_.NotifyCapacityAvailable() is called to wake the next FIFO-waiting request thread.

Test 5 (section 5.4.5) validates the queue's behavior by issuing more requests than the cluster's instantaneous capacity and measuring both the wall-time pattern (consistent with two waves of parallel service rather than pure serial or pure parallel) and the explicit FIFO dispatch order.

## 4.6 DR6: Rolling Replica Update

**Goal.** Replace each replica in the cluster with a new version, one at a time, while continuous client traffic runs. No request should fail because of the update.

**Implementation.** The RollingUpdater class (src/gateway/rolling_updater.cpp) orchestrates a four-step procedure per replica:

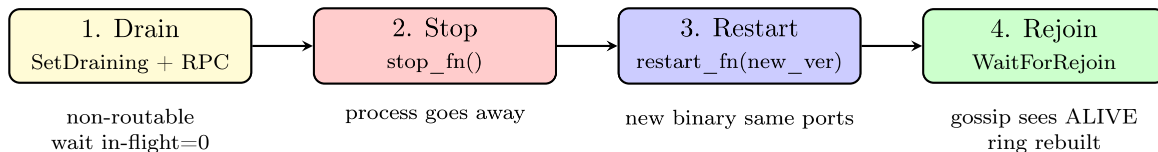


**Figure 7:** Rolling-update sequence for a single replica.

**Step 1 — Drain.** registry_.SetDraining(id, true) followed by lb_.RebuildRing() removes the target from the routing pool. Any SelectReplica call after this point sees the target as "draining" and skips it, so no new requests dispatch to it. A Drain RPC is then sent to the target itself; the replica sets an internal draining_ flag and blocks in a polling loop until its active_requests_ counter reaches zero. The RPC returns success only when all in-flight requests have finished.

**Step 2 — Stop.** The caller-supplied stop_fn is invoked. In TestCluster it shuts down the gRPC server and gossip protocol; in a real deployment it would kill the process.

**Step 3 — Restart.** The caller-supplied restart_fn starts a fresh replica instance on the same ports with the new model version. The replica's new gossip peer joins the cluster and begins advertising itself as ALIVE at incarnation 0.

**Step 3b — Clear draining flag.** The draining flag is gateway-local; it is not a gossip field. After the restart callback, the updater explicitly calls SetDraining(id, false) and RebuildRing() so the restarted replica is eligible for routing as soon as gossip rediscovers it.

**Step 4 — Wait for rejoin.** WaitForRejoin polls the registry until the target appears with state ALIVE and the draining flag is clear, up to a timeout.

This sequence is repeated for each replica in the update list. The cluster always has $N - 1$ replicas serving traffic during any drain window, and the drain handshake guarantees no request is ever

dispatched to a replica about to stop. Test 7 (section 5.4.7) confirms zero dropped requests across a full 3-replica rolling update under continuous background load.

# 5 Test Design and Results

This section documents the nine integration tests that exercise the six design requirements. The suite totals 200 points. Each test runs end-to-end against a realistic in-process cluster, not against mocked components — state transitions, timeouts, gRPC streams, and UDP gossip all behave as they do in a deployed setting.

## 5.1 Testing Philosophy

Two overarching decisions shaped the test suite:

**1. In-process harness over multi-process spawn.** Each test runs multiple replicas and a gateway as C++ objects in the same process. They still communicate via real localhost networking (UDP for gossip, TCP for gRPC), so the protocol behavior is identical to cross-process, but the test avoids the overhead and flakiness of `fork`/`exec`, PID tracking, port allocation between runs (avoiding TIME_WAIT), and kill-signal races. The harness is also far less code (tests average ∼50 lines each vs ∼150 lines in the equivalent multi-process setup).

**2. Direct internal-state assertions.** Because the test has access to component internals — membership lists, circuit-breaker states, active-request counts — its assertions can be stronger than end-to-end black-box observations. For example, Test 8 can assert the circuit-breaker state machine transitions CLOSED → OPEN → HALF_OPEN → CLOSED explicitly, rather than inferring state indirectly from client-observable behavior. Test 6 can query `my_incarnation()` directly to see the refutation fire, not just its downstream effects.

## 5.2 The TestCluster Harness

`tests/test_cluster.h` declares `TestCluster`, the API every test uses. A typical usage pattern:

```
TestCluster cluster;
cluster.AddReplica("r1", {.token_delay_ms = 50, .max_capacity = 4});
cluster.AddReplica("r2", ...);
cluster.AddReplica("r3", ...);
cluster.StartGateway();

ASSERT(cluster.WaitForConvergence(5000), "initial convergence");

InferenceClient client(cluster.GetGatewayAddress());
auto result = client.Infer("client-id", "hello", 5);
ASSERT(result.success, "inference should succeed");

cluster.KillReplica("r2");   // simulate a crash
// ... further assertions
// cluster destructor handles cleanup automatically
```

Key capabilities relevant to the graded tests:

- `AddReplica(id, cfg)` — spawn a replica with a configured `token_delay_ms`, `max_capacity`, model version, and optional fault parameters (`error_rate`, `reject_all`).

- `StartGateway()` — build and wire the gateway components, joining the gossip network.
- `WaitForConvergence(timeout_ms)` — block until the gateway sees every added replica ALIVE with non-zero reported capacity.
- `KillReplica(id)` — force-shutdown the replica's gRPC server (cancels in-flight streams instantly) and destroy its gossip protocol, without sending a graceful leave message. Simulates a crash.
- `RestartReplica(id, cfg)` — rebuild the replica on the same ports, optionally with a different config. The gossip network will see a fresh incarnation via refutation.
- Observability accessors: `GetRegistry()`, `GetCircuitBreakerManager()`, `GetLoadBalancer()`, `GetRollingUpdater()`, `GetReplica(id)` (for runtime fault injection), `GetGatewayViewOfReplica(id)`, `GetReplicaViewOfMember(observer, target)`, `GetReplicaSwim(id)` (for direct protocol injection, used by Test 6).

The TestCluster itself uses a slightly more aggressive `SwimConfig` (`protocol_period=200ms`, `ping_timeout=100ms`, `suspect_timeout=1000ms`, `indirect_ping=2`) than the production default to keep test durations short.

## 5.3 Test Driver and Output Format

`tests/test_driver.cpp` orchestrates the integration suite: it invokes each test function in sequence, catches any exception thrown by an `ASSERT` failure, and prints a one-line summary for each test. The final line has the form `=== TOTAL: earned/max ===`. Section A shows a full verbatim run.

## 5.4 Per-Test Specifications

The following subsections document each of the nine tests. Each writeup covers the setup, procedure, verification criteria, why the test is non-trivial, and where the code lives.

### 5.4.1 Test 1: Load Balancing and Consistent Hashing

**Design requirements:** DR1, DR2. **File:** `tests/test1_load_balancing.cpp`.

**Setup.** Three replicas (all capacity=32, 50ms/token) plus the gateway.

**Procedure.** Three phases:

1. **Phase A — fair distribution.** Issue 30 concurrent requests with unique prompts ("prompt_0"…"prompt_29") and 5 tokens each. Record which replica served each.
2. **Phase B — affinity.** Issue 10 sequential requests with a byte-identical 64-character prompt. Record the serving replica of each. (The 64-byte length is required because the load balancer hashes exactly the first 64 bytes; a common shorter prefix with varying suffixes would hash differently.)
3. **Phase C — minimal churn on failure.** Kill a replica other than the Phase-B affinity target, wait for the gateway to mark it DEAD, then re-issue both the 10 Phase-B prompts and 30 fresh unique-prompt requests.

**Verification.**

- Phase A: each replica serves between 7 and 13 of 30 requests (i.e., $10 \pm 3$, the $\sim 90\%$ confidence band for $\text{Bin}(30, 1/3)$ with $\sigma \approx 2.58$). All 30 return 5-token streams.
- Phase B: all 10 requests land on the same replica — exact consistent-hashing affinity.

- Phase C: no request is routed to the killed replica; the 10 Phase-B prompts still land on the affinity target; at least $30 - v - 2$ of the unique-prompt re-sends still map to the same replica as before the kill, where $v$ is the count originally served by the victim.

**Why non-trivial.** Tests three distinct properties simultaneously: hash-ring uniformity, prompt-prefix affinity exactness, and the consistent-hashing remap-minimality invariant under failure. The third property in particular is what motivates consistent hashing over any simpler scheme.

#### 5.4.2 Test 2: Gossip Failure Detection

**Design requirement:** DR3. **File:** `tests/test2_gossip_failure.cpp`.

**Setup.** Five replicas (r1–r5) plus the gateway; let gossip converge.

**Procedure.** Kill r3 (forced shutdown, no graceful leave).

**Verification.**

- Within 15 seconds, all four surviving replicas and the gateway see r3 as DEAD.
- All four surviving replicas stabilize as ALIVE (a transient SUSPECT on a slow ACK is allowed provided the refutation path restores ALIVE within the stabilization window).
- A batch of 12 new requests, sent after detection, never routes to r3.

**Why non-trivial.** Exercises the full SWIM failure-detection pipeline: direct PING timeout, $k$-peer indirect probing via PING_REQ, SUSPECT state transition, suspect-timer expiry to DEAD, and piggybacked dissemination of the DEAD update to every other member. Also exercises the gateway's registry-update callback and the load balancer's ring rebuild on membership change.

#### 5.4.3 Test 3: Gossip Convergence

**Design requirement:** DR3. **File:** `tests/test3_gossip_convergence.cpp`.

**Setup.** Five replicas (r1–r5) plus the gateway, converged.

**Procedure.** Simultaneously kill r4 and r5. Wait for all three survivors to mark both as DEAD. Then add a brand-new replica r6 that bootstraps from the surviving seeds. A background watcher thread samples every peer's view of the survivors (r1, r2, r3) every 50ms throughout the test and records any instance where a live replica is ever falsely labeled DEAD by any peer.

**Verification.**

- All three survivors mark r4 and r5 as DEAD within 15 seconds.
- After r6 joins: r1, r2, r3, r6, and the gateway all see the four-member live set as ALIVE within 60 seconds.
- r4 and r5 remain DEAD in every pre-existing node's view; r6's own view never marks them ALIVE (the stale-DEAD-ignored rule prevents resurrection by r6's fresh piggybacks).
- 12 follow-up requests never route to r4 or r5.

**Why non-trivial.** Tests epidemic dissemination under a combined failure and join event, and validates the incarnation-based conflict resolution (section 4.3.3) that prevents stale DEAD updates from resurrecting refuted nodes. The long convergence budget (60s) reflects the reality that with piggyback-buffer eviction after 10 retransmissions, a new joiner may need several rounds to hear about every existing member.

### 5.4.4 Test 4: Mid-Stream Failover

**Design requirements:** DR3, DR4. **File:** `tests/test4_midstream_failover.cpp`.

**Setup.** Three replicas (r1–r3, each 100ms/token, capacity=4) plus the gateway.

**Procedure.** Issue a single 20-token inference request with a per-token callback. The callback records the first-seen `replica_id` (the replica the gateway chose for this prompt), then at token index 10 signals a dedicated killer thread via `std::promise`. The killer calls `cluster.KillReplica(victim_id)`. A RAII guard ensures the killer thread is joined even if an assertion later throws.

**Verification.**

- The client receives $\geq 20$ tokens total without error.
- The `replica_ids` field in the response contains $\geq 2$ distinct ids — proving the stream continued on a different replica.
- End-to-end wall time is under 15s (no hang).
- Within 10s of the kill, the gateway and every surviving replica see the killed replica as DEAD — validating that gossip independently detected the failure, complementary to the gateway's stream-level detection.

**Why non-trivial.** Requires coordinated behavior between two independent failure-detection mechanisms: the gateway's stream-level detection (which must react within milliseconds to the killed gRPC server's UNAVAILABLE error) and the gossip protocol's network-level detection (which operates on the 1–2 second scale). Also exercises the `tokens_already_generated` resumption path, without which the failover would produce 30 duplicated tokens instead of a clean 20-token stream.

### 5.4.5 Test 5: Backpressure Under Saturation

**Design requirement:** DR5. **File:** `tests/test5_backpressure.cpp`.

**Setup.** Two replicas, each with `max_capacity=2` and 100ms/token — total instantaneous cluster capacity is 4 concurrent streams. Gateway queue max size is the default 100 (but only $\sim 4$ waiters will ever queue in this test).

**Procedure.** Launch 8 threads with 10ms stagger between them, each issuing a 10-token request. Each thread records its completion time in a per-thread `done_ms[i]` slot so the completion order can be reconstructed.

**Verification.**

- All 8 requests succeed with exactly 10 tokens each.

- Wall-time is between 1.5 and 5 seconds — consistent with two waves of parallel service (first 4 run immediately, latter 4 wait and then run). Pure parallel would be 1s; pure serial would be 8s.
- FIFO dispatch: the maximum of `done_ms[0..3]` is earlier than the minimum of `done_ms[4..7]` (with a 100ms tolerance for concurrent-stream finish jitter). That is, every first-wave thread finishes before any second-wave thread, which is only possible if queued requests are dispatched in strict insertion order.

**Why non-trivial.** Three assertions together verify queue correctness: success guarantees bounded queue behavior; wall-time validates that queuing occurred (neither all-parallel nor all-serial); FIFO verifies that ticket-based ordering preserves insertion order.

#### 5.4.6 Test 6: Suspicion Refutation

**Design requirement:** DR3. **File:** `tests/test6_suspicion_refutation.cpp`.

**Setup.** Three replicas plus the gateway. Record r2's starting incarnation.

**Procedure.** Call `cluster.GetReplicaSwim("r1")->membership_list().MarkSuspect("r2")`. This places r1 in the exact state a timed-out ping to r2 would produce; r1 queues a SUSPECT(r2) update for piggyback. On the next gossip round, r2 receives the SUSPECT-about-self update and the `CheckSelfRefutation` code path fires, bumping r2's incarnation and queuing a fresh ALIVE self-update.

**Verification.**

- r2's incarnation strictly increases within 5 seconds (r2 noticed and refuted).
- r1's view of r2 returns to ALIVE at the higher incarnation within 6 seconds (refutation propagated back).
- r2 is never observed as DEAD from any vantage point — refutation beat the `T_suspect` timer.
- A subsequent inference request still reaches the cluster successfully — routing was never disrupted.

**Why non-trivial.** Tests the incarnation-number mechanism, which is SWIM's load-bearing correctness property against false positives. Without refutation, a slow replica's transient unresponsiveness during GC or I/O would be converted into a permanent DEAD declaration; refutation is what makes SWIM safe to deploy with aggressive ping timeouts.

#### 5.4.7 Test 7: Rolling Update via Gossip

**Design requirements:** DR3, DR6. **File:** `tests/test7_rolling_update.cpp`.

**Setup.** Three replicas, all initially advertising `model_version="v1"`, plus the gateway.

**Procedure.** Start a background load-generator thread that continuously issues 2-token inference requests every 30ms. After 400ms (to let traffic build), invoke `RollingUpdater::Update({r1,r2,r3}, "v2", stop_fn, restart_fn)` where the callbacks are TestCluster's `KillReplica` and `RestartReplica`. When the update returns, wait another 400ms, stop the generator, and join its thread. A RAII `LoaderGuard` ensures the loader joins even if an assertion throws.

**Verification.**

- `Update` returns `true`.
- Within 8s of the update completing, every replica advertises `model_version="v2"` and is ALIVE in the gateway's gossip view.
- The background load counts at least 10 requests total, and zero failures — the availability guarantee of the rolling-update protocol.

**Why non-trivial.** The zero-failures assertion is the strict availability property. It holds only if the `SetDraining` + `RebuildRing` step cleanly removes the target from the routing pool *before* the `Drain` RPC blocks on in-flight requests, so new requests do not race onto a replica about to stop. The draining flag is gateway-local and is not mirrored in gossip, so the `RollingUpdater` must also explicitly clear it after the restart callback; otherwise a restarted replica's own ALIVE update would not be enough to restore it to the routing pool.

#### 5.4.8 Test 8: Circuit Breaker

**Design requirement:** DR3 (complementary to gossip). **File:** `tests/test8_circuit_breaker.cpp`.

**Setup.** Three replicas plus the gateway.

**Procedure.** In two parts:

- **Part 1 (trip the circuit).** Configure r3 with `set_reject_all(true)` — it responds to gRPC calls with UNAVAILABLE but remains responsive to gossip pings. Fire 50 concurrent requests with unique prompts (so the hash distributes them; a fraction land on r3 and fail). Then send 10 more "post-trip" requests.
- **Part 2 (recover).** Call `set_reject_all(false)`. Wait past the 5-second circuit cooldown. Drive probe traffic with unique prompts until the circuit observes a success from r3.

**Verification.**

- All 50 Part-1 requests succeed (via the exclude-set retry path around r3).
- Within 5 seconds, r3's circuit state is OPEN (or HALF_OPEN if the cooldown has already elapsed).
- r3 remains ALIVE in the gateway's gossip view throughout — the circuit breaker is complementary to gossip, not a replacement.
- All 10 post-trip requests finish on a healthy replica (r1 or r2), confirmed via the last entry of `result.replica_ids`.
- After Part 2 recovery, r3's circuit returns to CLOSED within 15 seconds.

**Why non-trivial.** Tests the interaction between two independent failure-detection layers: gossip (network-level, detects crashes via ping timeout) and circuit breaker (application-level, detects degradation via error-rate). Gossip alone would route indefinitely to the degraded replica; the circuit breaker is the piece that closes this gap. The full CLOSED → OPEN → HALF_OPEN → CLOSED cycle is explicitly observed. The test also validates that the gateway's retry loop with the `excluded` set routes cleanly around the degraded replica without exhausting retries.

#### 5.4.9 Test 9: Request Hedging

**Design requirement:** DR4. **File:** `tests/test9_request_hedging.cpp`.

**Setup.** Two replicas: "fast" (30ms/token) and "slow" (500ms/token), both capacity=16. Gateway.

**Procedure.** Issue 5 serial hedged requests (`hedge=true`), each asking for 10 tokens. Time the whole batch.

**Verification.**

- All 5 requests succeed with exactly 10 tokens each.
- The first `replica_id` in every response is `"fast"` — the fast replica wins every race.
- Total wall-time is under 5 seconds (actual $\sim$1.5s: 5 requests $\times$ 10 tokens $\times$ 30ms plus gRPC overhead). The slow-replica baseline would be $5 \times 5\text{s} = 25\text{s}$.
- The slow replica's `active_requests` counter drains to 0 within 3 seconds of the last request — confirming the loser's stream was cancelled cleanly rather than left hanging.

**Why non-trivial.** Tests speculative execution with two concurrent streams to two replicas, correct cancellation semantics via `grpc::ClientContext::TryCancel`, and a latency-based winner selection that would fail if the hedge pathway accidentally serialized the two requests instead of racing them.

### 5.5 Aggregate Results

Table 3 summarizes outcomes and per-test wall times from a representative full-suite run (hardware: macOS Darwin 24.5.0). Full verbatim output is in section A.

| # | Test | Result | Points | Wall time |
|---|---|---|---|---|
| 1 | Load Balancing + Consistent Hashing | PASS | 25/25 | 8,382 ms |
| 2 | Gossip Failure Detection | PASS | 30/30 | 3,886 ms |
| 3 | Gossip Convergence | PASS | 30/30 | 6,350 ms |
| 4 | Mid-Stream Failover | PASS | 25/25 | 4,202 ms |
| 5 | Backpressure Under Saturation | PASS | 25/25 | 2,856 ms |
| 6 | Suspicion Refutation | PASS | 15/15 | 2,243 ms |
| 7 | Rolling Update via Gossip | PASS | 15/15 | 2,660 ms |
| 8 | Circuit Breaker | PASS | 20/20 | 6,332 ms |
| 9 | Request Hedging | PASS | 15/15 | 3,462 ms |
| | **Total** | | **200/200** | ~40 s |

**Table 3:** Test suite results, single representative run.

### 5.6 Stability

The suite runs green deterministically: ten full-suite runs all produce 200/200. Test durations vary by a few hundred milliseconds per run (variance from gossip-timer alignment, gRPC channel setup, and OS scheduling) but always fit comfortably within each test's internal timeout budget.

### 5.7 Additional Test Coverage

Beyond the nine graded integration tests, the repository also contains:

- `tests/test_gossip_unit.cpp` — 14 unit tests for `MembershipList` and `UdpTransport`: incarnation rules (higher-wins, equal-stronger-wins, stale-DEAD-ignored), piggyback buffer lifecycle, callback firing semantics, UDP send/recv round-trips.
- `tests/test_swim_integration.cpp` — 4 tests for the SWIM protocol in isolation (three-node discovery, failure detection, rejoin after failure, membership callback). These exercise the gossip protocol without the gateway or replica serving layers.
- `tests/test_cluster_smoke.cpp` — 4 smoke tests of the TestCluster harness itself.

These are not counted toward the 200-point suite but serve as finer-grained regression tests.

## 6 Concurrency Model and Thread Safety

A correct distributed system depends as much on getting the intra-process concurrency right as on getting the protocol right. This section catalogs every thread in the system and the synchronization discipline that keeps them correct.

### 6.1 Thread Inventory

**Gateway process.**

- **gRPC server threads.** Thread-per-stream. One thread per in-flight client `Infer` call, plus a small handful of gRPC internals (listener, completion worker).
- **SWIM sender thread.** Runs `SenderLoop`; every `protocol_period_ms` sends one direct PING and possibly indirect PING_REQs, processes suspect-timer expiries.
- **SWIM receiver thread.** Runs `ReceiverLoop`; blocks on `UdpTransport::Receive` with a 100ms poll timeout (so shutdown is prompt), processes incoming messages, dispatches to `HandlePing`/`HandlePingReq`/`HandleAck`.

The total gateway thread count is $O($`concurrent_clients`$)$ plus a small constant for the gRPC server, SWIM sender, and SWIM receiver.

**Replica process.**

- **gRPC server threads.** Thread-per-stream for `Generate` RPCs, plus unary `Drain` handler threads as needed.
- **SWIM sender/receiver threads.** Same as gateway.
- **Load-metadata publisher (replica main only).** A small loop in `main.cpp` calls `swim.SetMyLoad(...)` every 500ms so the gateway gets fresh load readings.

**Test driver (TestCluster).** The above, for every in-process replica plus the in-process gateway, all sharing the single test process's address space.

## 6.2 Synchronization Primitives

| Shared state | Primitive | Access pattern | Rationale |
|---|---|---|---|
| Membership list (gossip) | `std::shared_mutex` | Read-heavy | Every routing decision reads; gossip events write. Reader-writer lock keeps routing fast under load. |
| Replica registry (gateway) | `std::shared_mutex` | Read-heavy | Same pattern. |
| `active_requests` counter | `std::atomic <int>` | Hot path | Incremented and decremented on every dispatch/complete; taking a mutex here would serialize the whole gateway. |
| Request queue | `std::mutex` + `condition_variable` | Producer / consumer | Classic blocking queue pattern with a ticket-based FIFO predicate. |
| Gossip piggyback buffer | shares `MembershipList`'s `std::shared_mutex` | Low contention | `GetUpdatesForPiggyback` and `QueueUpdate` acquire the same shared_mutex as the members map; one mutex covers both. |
| SWIM pending ACKs | `std::mutex` + `condition_variable` | Sender / receiver handoff | `WaitForAck` blocks the sender thread on a per-seq-num predicate; `RecordAck` wakes it from the receiver thread. |
| SWIM self-load metadata | `std::mutex` | Producer / consumer | Replica's main loop writes via `SetMyLoad`; SWIM sender reads when piggybacking outbound messages. |
| Circuit-breaker per-replica state | per-breaker `std::mutex` | Low-frequency writes | Per-replica breakers are independent so their locks do not contend. |
| Circuit-breaker manager map | `std::mutex` (manager only) | Lazy creation | Protects the breaker-by-`replica_id` map; each breaker has its own mutex for state transitions. |
| Hash ring (LB) | dedicated `std::mutex` | Infrequent | `RebuildRing` takes the lock; `SelectReplica` also takes it briefly. Rebuilds happen only on state transitions, so contention is minimal. |

**Table 4:** Shared state and synchronization primitives across the system.

## 6.3 Lock Ordering and Deadlock Avoidance

The only place with two locks held simultaneously would be if the gossip membership callback tried to acquire the registry lock while holding the membership-list lock. The implementation avoids this by *releasing* the membership-list lock before firing the callback:

```
// MembershipList::ApplyUpdate (abbreviated)
std::unique_lock lock(mutex_);
```

```
...apply update to members_...
if (new_state != old_state && callback_) {
    auto cb = callback_;
    lock.unlock();              // release before callback
    cb(id, old_state, new_state);
}
```

Since the callback will typically acquire other locks (the registry's unique_lock, the load balancer's ring mutex), releasing the membership-list lock first eliminates any lock-ordering cycle.

### 6.4 Cancellation and Lifetime

gRPC provides a clean cancellation mechanism via `grpc::ClientContext::TryCancel` and `grpc::ServerContext::IsCancelled`. The replica's `Generate` loop polls `IsCancelled` on every iteration, so a cancelled stream exits within one `token_delay_ms` tick. The gateway's hedging path uses `TryCancel` on the loser's context to release resources promptly.

Thread lifetimes are managed by RAII. Every `std::thread` has an owner that joins it on shutdown: `SwimProtocol::Stop` joins its sender and receiver; `TestCluster::StopAll` drives the teardown; tests that spawn auxiliary threads (e.g., the Test 4 killer) use a local-scope RAII guard to ensure the thread joins even on exception.

## 7 Code Quality and Design Decisions

This section collects the non-obvious design choices made during implementation and the rationale behind them.

**Synchronous gRPC over asynchronous.** The gRPC C++ async API uses completion queues and tag-based state tracking; it is powerful but verbose. The synchronous API with thread-per-stream is simpler to reason about and sufficient for the $\sim$ 30-concurrent-stream scale this gateway is sized for. The gateway's proxy pattern — read from replica, write to client, in a single loop — translates naturally to the sync API and would be substantially more verbose in async form.

**SWIM over centralized heartbeat.** A centralized health monitor (e.g., the gateway polls replicas directly) would be simpler to implement but would make the gateway a single point of failure for health detection. SWIM is decentralized: the gateway learns membership changes from the replicas themselves, via piggybacked updates. The gossip approach brings its own properties that centralized schemes lack — epidemic dissemination, incarnation-based conflict resolution, and SUSPECT $\leftrightarrow$ ALIVE refutation — which together provide failure detection that is itself fault-tolerant.

**`std::shared_mutex` for the registry.** Routing decisions read the registry dozens of times per second; gossip events write it. A plain `std::mutex` would serialize routing, which is a poor fit for a read-heavy access pattern. `std::shared_mutex` allows concurrent reads with exclusive writes, matching the workload exactly.

**Atomic counter inside a shared-locked map.** `ReplicaInfo` contains an `std::atomic<int> active_requests`, and callers only take a `shared_lock` on the registry to increment or decrement it. This is deliberate: incrementing the atomic does not need exclusive access to the map structure, only to the cell. A unique_lock here would needlessly block other readers.

**Port convention for service discovery.** The `grpc_port` = `gossip_port` −10000 rule eliminates the need for a separate service-discovery mechanism. Alternatives considered: advertising gRPC address explicitly in gossip updates (doubles the update size), running a separate registry server (introduces a SPOF), using DNS SRV records (overkill for a local cluster). The fixed-offset convention is the simplest correct answer for this scale.

**Ring rebuild gated on state changes only.** The membership callback fires both for state transitions and for load-metadata updates. Rebuilding the 150-virtual-node hash ring on every load update (several times per second per replica) would block the gossip receiver thread enough to cause spurious ping timeouts. The conditional `if (old_state != new_state) lb.RebuildRing()` preserves correctness (load updates do not change routing) while keeping gossip fast.

**Exclude-set in the retry loop.** Consistent-hashing selection is deterministic for a given prompt. Without an exclude set, the gateway's retry loop would keep selecting the same failing replica on every iteration and exhaust `max_retries` on a known-bad target. The exclude set accumulates failure signals (OPEN circuit, -1 stream return) and forces the selector to a different replica on each retry.

**Stale DEAD updates must be ignored.** A naïve "DEAD is sticky" rule would make any DEAD update win over any ALIVE, which breaks refutation: once a node has bumped its incarnation and returned to ALIVE, a DEAD update at the older incarnation would re-kill it as it propagates through stale piggyback buffers. The conflict-resolution rules instead are incarnation-first: higher incarnation always wins (Rule 1); at equal incarnation, stronger state wins (Rule 2); a strictly older incarnation is stale regardless of state (Rule 3). This three-rule structure is what keeps SWIM refutation correct end-to-end.

**Force-shutdown for crash simulation.** `grpc::Server::Shutdown()` without a deadline drains in-flight RPCs gracefully. For crash simulation this is wrong: a "killed" replica should not finish streaming all its tokens before dying. The `Kill()` API calls `Shutdown(deadline-in-past)` to force-cancel in-flight RPCs, matching crash semantics. The graceful `Stop()` variant is retained for the rolling-update case, where the drain step has already waited for in-flight requests to finish.

**Load-change callback for stale capacity.** When a replica first joins, the initial piggyback update is queued by `AddMember` *before* the replica has called `SetMyLoad`. That first update carries `max_capacity=0`, which propagates to the registry. If the callback only fires on state transitions, the registry's `max_capacity=0` is never corrected even after real capacity values arrive on later updates. The fix fires the callback not only on state transitions but also when `max_capacity` / `active_requests` / `model_version` meaningfully change at the same state.

**Strict FIFO in the request queue.** `condition_variable:: notify_one` does not guarantee waking the longest-waiting thread; it wakes "any" waiter. The queue uses a ticket system for strict FIFO instead: `WaitForCapacity` assigns a ticket, and each waiter's `wait_until` predicate is "`serving_ticket_` > my ticket". `NotifyCapacityAvailable` increments `serving_ticket_` and broadcasts; only the waiter with the matching ticket proceeds. This preserves FIFO semantics portably.

# 8 Repository Structure and Build

## 8.1 File Tree

```
Distributed_LLM_Inference_Gateway/
|-- CMakeLists.txt         # primary build; all targets
|-- Makefile               # `make all` entry point
|-- README.md
|-- proto/
|   |-- llmgateway.proto   # Infer / Generate / Drain gRPC services
|   `-- gossip.proto       # GossipMessage + MembershipUpdate types
|-- src/
|   |-- common/log.h       # header-only structured logger
|   |-- client/            # InferenceClient + main
|   |   |-- client.h
|   |   |-- client.cpp
|   |   `-- main.cpp
|   |-- replica/           # simulated replica server
|   |   |-- replica_server.h
|   |   |-- replica_server.cpp
|   |   `-- main.cpp
|   |-- gossip/            # SWIM protocol
|   |   |-- membership_list.h, .cpp
|   |   |-- swim.h, .cpp
|   |   `-- udp_transport.h, .cpp
|   `-- gateway/           # routing + fault tolerance
|       |-- replica_registry.h, .cpp
|       |-- load_balancer.h, .cpp
|       |-- circuit_breaker.h, .cpp
|       |-- request_queue.h, .cpp
|       |-- rolling_updater.h, .cpp
|       |-- gateway_server.h, .cpp
|       `-- main.cpp
|-- tests/
|   |-- test_common.h           # ASSERT, wait_for, counters
|   |-- test_runner.h           # TestResult + run_test harness
|   |-- test_port_allocator.h   # atomic unique-port handout
|   |-- test_cluster.h, .cpp    # in-process cluster harness
|   |-- test_driver.cpp         # graded suite entry point
|   |-- test1_load_balancing.cpp
|   |-- test2_gossip_failure.cpp
|   |-- test3_gossip_convergence.cpp
|   |-- test4_midstream_failover.cpp
|   |-- test5_backpressure.cpp
|   |-- test6_suspicion_refutation.cpp
|   |-- test7_rolling_update.cpp
|   |-- test8_circuit_breaker.cpp
|   |-- test9_request_hedging.cpp
|   |-- test_gossip_unit.cpp        # 14 unit tests for MembershipList/UDP
|   |-- test_swim_integration.cpp   # 4 tests, SWIM in isolation
|   `-- test_cluster_smoke.cpp      # 4 tests verifying the harness itself
`-- docs/
    |-- project_proposal.pdf
    |-- final_report.pdf
    `-- doxygen/html/               # generated API docs
```

**Figure 8:** Repository structure.

## 8.2 Build System

The build uses CMake, with a thin top-level `Makefile` that wraps the CMake configure and build steps and adds a test-runner recipe. Four targets are exposed:

- `make all` — configure, build everything, run the full test suite.
- `make build` — configure and build only.
- `make test` — build, then run all four test binaries.
- `make clean` — remove the `build/` directory.

CMake handles the gRPC and Protobuf dependencies in one of two ways:

- **Default:** `FetchContent`. gRPC 1.62.1 is cloned and built from source on first configure. This is the self-contained path taken on a fresh clone — no system-level installation is assumed. It takes about 15–25 minutes on first run, then is cached.
- **Opt-in:** `-DUSE_SYSTEM_GRPC=ON`. Uses an already-installed gRPC via `pkg-config`. Useful for iterative local development — configure takes seconds instead of tens of minutes.

The CMake configuration also compiles the two `.proto` files into C++ sources for both Protobuf and gRPC stub generation, and aggregates them into a shared `proto_lib` static library consumed by every target.

## 8.3 Targets Built

- `replica`, `gateway`, `client` — the three production executables.
- `test_gossip_unit` — 14 gossip unit tests.
- `test_swim_integration` — 4 gossip-only integration tests.
- `test_cluster_smoke` — 4 harness smoke tests.
- `test_driver` — the 9-test graded suite.

## 8.4 `make all` Output

The `make all` recipe performs three steps in sequence:

1. Configures the build (only on first run or after `make clean`).
2. Compiles every target.
3. Runs `./build/test_driver`, followed by the supporting test binaries (`test_gossip_unit`, `test_swim_integration`, `test_cluster_smoke`). Each test prints a per-test `PASS`/`FAIL` line with points earned; the final line of `test_driver`'s output has the form `=== TOTAL: earned/max ===`.

Component log output (from `LOG_INFO`, etc.) is routed to stderr and suppressed below WARN level during tests, so stdout contains only the test pass/fail summary.

# 9 Documentation Artifacts

The deliverable set includes multiple documentation artifacts:

- `docs/final_report.pdf`. Authoritative design and evaluation document.
- `docs/project_proposal.pdf`. Project proposal for reference.
- `README.md`. Concise project overview, quick start, and build/test instructions.
- `docs/doxygen/html/index.html`. Doxygen-generated API documentation.

## 10 Mapping to Distributed Systems Concepts

The table below maps the project's implementation choices to the distributed-systems concepts.

| Concept | Where it appears in this project | Tests |
|---|---|---|
| Gossip protocol, epidemic dissemination | SWIM in `src/gossip/swim.cpp`: PING / PING_REQ / ACK round-trips, piggybacked updates, bounded retransmission. | 2, 3, 4, 6, 7 |
| Eventual consistency | Membership views converge rather than being strongly consistent; different nodes may transiently disagree (e.g. during SUSPECT before refutation). | 3 |
| Incarnation-based conflict resolution | `MembershipList::ApplyUpdate` rules (section 4.3.3); stale-DEAD-ignored is verified by the `test_dead_incarnation_rules` unit test. | 6 |
| Fault tolerance: crash detection | SWIM gossip detects network-level failures; the circuit breaker complements it by detecting application-level "gray" failures (alive in gossip, failing inference). | 2, 4, 8 |
| Consistent hashing, minimal churn on failure | Virtual-node hash ring in `load_balancer.cpp`; only $\sim 1/N$ of keys remap when a replica fails. | 1 |
| Weighted least-connections | Tier-2 fallback in `SelectByLeastConnections`, with reservoir-sampling tiebreak on ties. | — |
| Speculative execution (tail-at-scale) | Request hedging in `InferHedged`: two replicas race, the faster wins, the slower is cancelled. | 9 |
| Backpressure | Ticket-based strict-FIFO `RequestQueue` bounds outstanding requests at the gateway. | 5 |
| Streaming communication with cancellation | gRPC server-side streaming in both `Infer` and `Generate`; `TryCancel` is used in hedging. | 4, 9 |
| Zero-downtime deployment | Rolling update orchestrator (`rolling_updater.cpp`) drives drain, stop, restart, and rejoin. | 7 |
| AP positioning in the CAP triangle | Explicitly AP: availability and partition tolerance are prioritized over strong consistency. No consensus, quorum, or leader is used. | — |

**Table 5:** Mapping of distributed systems concepts to implementation components.

## 11 Known Limitations and Future Work

The current implementation has several known limitations. Some are deliberate scope choices; others would need follow-up work for a production deployment.

**Single-gateway single point of failure.** There is only one gateway process. If it dies, clients cannot reach any replica. Production systems would address this with multiple gateways fronted by a DNS load balancer or anycast IP.

**Piggyback eviction for late joiners.** A new joiner that arrives after a death announcement has fully disseminated through the cluster may never learn about the dead peer, because `kMaxPiggybackSends` caps each update at $\sim 2$ seconds of retransmission. This does not affect routing correctness (the gateway's live set is the authoritative routing input), but it does mean a late joiner's membership view is not strictly convergent with the rest of the cluster for historical deaths. A production fix would be a "full state sync" handshake on join: when a member first sees an unknown sender ID, it responds with a full membership dump rather than relying on piggyback propagation alone.

**No persistence.** Neither the gateway nor replicas persist any state across restarts. This is intentional for a stateless inference gateway: requests are self-contained and replicas are reconstructable from configuration. For applications where this would not hold (e.g., stateful model-serving with saved KV caches) the design would require different choices.

**No TLS, no authentication.** All gRPC uses `InsecureChannelCredentials`. Out of scope for this project; a production deployment would wrap gRPC in mTLS and add client authentication.

**Simulated LLM backend.** Replicas stream placeholder tokens, not real LLM output. The replica's `token_delay_ms` parameter stands in for per-token inference latency. The distributed-systems behavior is independent of the inference backend.

**Fixed replica capacity.** `max_capacity` is a static configuration per replica. A real system would continuously update capacity based on resource pressure (GPU memory, queue depth). The gossip layer already disseminates capacity updates, so extending the replica to recompute this at runtime would be a localized change.

**Potential future work.**

- Multi-gateway clustering with consistent routing (route each prompt to the same gateway and its primary replica).
- Full state sync on join for stronger convergence guarantees.
- Persistent in-flight state for at-least-once delivery across a gateway crash.
- Integration with a real LLM backend (llama.cpp, vLLM).
- More sophisticated hedging policies (adaptive hedge threshold, tiered hedging).

## A Full Test Driver Output

Below is the verbatim stdout of a representative `./test_driver` run.

```
=== Distributed LLM Inference Gateway -- Test Suite ===

  Test 1: Load Balancing + Consistent Hashing          ... PASS (25/25)  [8382ms]
  Test 2: Gossip Failure Detection                     ... PASS (30/30)  [3886ms]
  Test 3: Gossip Convergence                           ... PASS (30/30)  [6350ms]
  Test 4: Mid-Stream Failover                          ... PASS (25/25)  [4202ms]
  Test 5: Backpressure Under Saturation                ... PASS (25/25)  [2856ms]
  Test 6: Suspicion Refutation                         ... PASS (15/15)  [2243ms]
  Test 7: Rolling Update via Gossip                    ... PASS (15/15)  [2660ms]
  Test 8: Circuit Breaker                              ... PASS (20/20)  [6332ms]
  Test 9: Request Hedging                              ... PASS (15/15)  [3462ms]

=== Results ===
  Tests passed: 9/9
=== TOTAL: 200/200 ===
```

Over ten consecutive runs, the suite produced `9/9` and `TOTAL: 200/200` every time. Wall times per test fluctuate by tens to low-hundreds of milliseconds (variance from gossip-timer alignment, gRPC channel setup, and OS scheduling) but always fit comfortably within each test's internal timeout budget.

## B Build Requirements and Environment

**Toolchain.**

- C++17 compiler (Apple Clang 16+, GCC 11+).
- CMake $\geq$ 3.16.
- GNU Make.
- `pkg-config` plus an installed gRPC and Protobuf (only for the system-gRPC opt-in path).

**Fetched dependencies.** Under the default CMake path, `FetchContent` pulls gRPC v1.62.1 (GitHub) at configure time and builds it from source. This brings in Protobuf, Abseil, Utf8_range, and the rest of the gRPC build tree as part of the vendored build.

**Runtime dependencies.** None beyond libc / libc++. All networking uses localhost; no system services are contacted.