Distributed Canary Deployment – Design Plan

Selin Demirtürk, Roberto Villafuerte, Jan Thurner

November 14, 2025

Contents

1 Introduction

We design a distributed Canary Model Rollout Orchestrator for ML services at global scale. Production currently serves model `v1`. A new model `v2` is introduced gradually (e.g., 5% → 10% → 50% → 100%) only if health is good. A tiny global routing state (`{model_id -> weight, version, status}`) is agreed atomically across all serving nodes so that every region flips to the same configuration at the same logical moment. If latency/error SLOs degrade, the system aborts the change or rolls back to the last good version. The full-scale vision can be used for multiple regions with hundreds of nodes per region, each region makes local routing decision using the same protocol.

Top-level Architecture

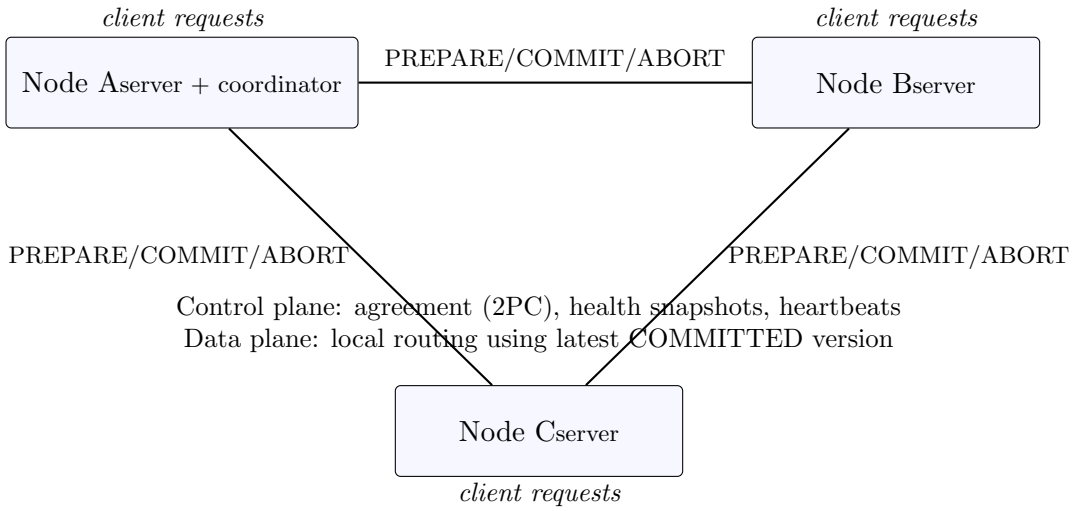

Figure 1: Three-node prototype (scales to N). Each node talks to the other two.

2 Topics and Techniques

We implement a distributed *canary rollout orchestrator* where all serving nodes agree on a tiny, versioned `RoutingState` ({model→weight, version, status}). A change (e.g., 5%→10%) is published via **Two-Phase Commit (2PC)**: during PREPARE each node first *stages* the canary artifact (verify checksum, warm-load the model), runs quick *smoke tests* and local health gates, and persists `PREPARED`. If every node votes `COMMIT`, the coordinator broadcasts GLOBAL_COMMIT; each node atomically flips its in-memory router to the new weights, starts serving per those weights, and persists `COMMITTED`. Otherwise, GLOBAL_ABORT keeps the previous version. Messages are idempotent (carry `txid`, `version`); state transitions are durably logged and recovered by replay/peer queries, preventing mixed views after crashes or partitions.

2.1 Global Routing State

We maintain a single source of truth replicated on all nodes:

Listing 1: RoutingState (persisted and agreed cluster-wide).

```
{
  "version": 18,                          // monotonically increasing
  "stable_model_id": "v1",
  "canary_model_id": "v2",
  "weights": { "v1": 0.95, "v2": 0.05 },  // sum = 1.0
  "status": "PREPARED|COMMITTED|ABORTED",
  "txid": "promote-2025-11-12T10:22:31Z"
}
```

2.2 Consistency and Consensus (2PC)

We use **Two-Phase Commit** to make state changes atomic:

1. **Phase 1: PREPARE.** Coordinator proposes next version (e.g., 5%→10%). Each participant: warm-loads model, checks health gates, persists `PREPARED <version>`, then votes `COMMIT` or `ABORT`.
2. **Phase 2: DECISION.** If all votes are `COMMIT`, coordinator broadcasts `COMMIT <version>`; else broadcasts `ABORT <version>`. Participants atomically apply the decision and persist it.

Stretch goal: Replace 2PC with **Raft** over a change log to allow majority progress under certain failures.

2.3 Operational Parameters (SLOs, Timeouts, Ports)

- **Health gates (window =** $\min\{60\ \mathbf{s}, 2000\ \mathbf{reqs}\}$**)**: $p95_{\text{lat}} \leq 1.2 \times p95_{\text{baseline}}$, error_rate $\leq 0.5\%$.
- **2PC timeouts**: PREPARE reply timeout = 2 s; decision timeout = 1 s; backoff jitter 100–300 ms; max retries = 3.
- **Ports**: Node A: 50051; Node B: 50052; Node C: 50053.
- **Snapshots**: every 5 committed versions; retain last 50 log entries.

2.4 Failure Modes & Recovery

2.4.1 Coordinator crash during PREPARE

Participants time out, elect a temporary resolver (highest node id), query peers for decision: commit if any peer has `COMMITTED <v>`, else `ABORT <v>`. Guarantees no mixed views.

2.4.2 Participant crash before vote

Coordinator times out and decides `ABORT`. Crashed node replays log on restart and serves last COMMITTED version.

2.4.3 Participant crash after PREPARED

Coordinator may still `ABORT`; on restart, node finds `PREPARED <v>` without decision, queries peers, applies final decision.

2.4.4 Network partition

2PC requires unanimous votes; promotion pauses or aborts under partition. Already committed versions remain; nodes catch up via heartbeat upon heal.

2.5 Persistence and Recovery

Each node keeps an append-only JSON log of state transitions and periodically snapshots the last COMMITTED state. On restart, a node replays its log to reconstruct state. If it finds a highest `PREPARED` version without decision, it queries peers: commit if any peer has `COMMITTED`, otherwise abort. Heartbeats exchange `{node_id, last_committed_version, digest}` for catch-up.

2.6 Scalability Considerations

The control plane manipulates a tiny object; its cost grows slowly with node count. The data plane scales horizontally by adding serving replicas. Global deployment can be organized per-region with the same protocol, then orchestrated region-by-region for staged rollouts.

3 Nodes and their Roles

3.1 Common

- **Serving**: `/predict` routes requests by local `weights` of the latest COMMITTED version (optional sticky sessions).
- **State Manager**: Holds in-memory `RoutingState`; persists `PREPARED/COMMITTED/ABORTED`; snapshots; replay on startup.
- **Health Watcher**: Maintains rolling metrics; exposes `/health/snapshot`.
- **Peer Comms**: Sends/receives heartbeats; requests missing decisions to catch up.

3.2 Coordinator

- Proposes next version (e.g., 5%→10%) and runs 2PC.
- Applies timeouts; if a participant is unresponsive, decides `ABORT` (safety first).
- Triggers rollback (proposes prior stable version) if SLO breach is detected.

3.3 Participants

- On `PREPARE`: warm-load, check gates, persist `PREPARED`, vote.
- On decision: atomically swap weights; persist `COMMITTED`/`ABORTED`.

4 Messaging

4.1 Control Plane (peer ↔ peer)

Listing 2: Core control messages.

```
message RoutingState {
  int64  version;
  string stable_model_id;
  string canary_model_id;
  // weights serialized as "model_id:share" pairs
  repeated string weights_k;    // e.g., "v1"
  repeated double weights_v;    // e.g., 0.95
  string status;                // PREPARED|COMMITTED|ABORTED
  string txid;
}

message PrepareReq  { int64 version; RoutingState state; string txid; }
message PrepareResp { string txid; enum Vote { COMMIT=0; ABORT=1; } vote;
                                                          string reason; }

message Decision    { int64 version; string txid; enum Kind { COMMIT=0; ABORT=1;}
                                                                    kind; }
message Ack         { string txid; bool ok; }

message HealthSnap  { string node_id; double p95_ms; double err_rate; int64 n;
                                                          string window_id; }
message Heartbeat   { string node_id; int64 last_committed_version; bytes digest;
```

4.2 Data Plane (client → server)

Listing 3: HTTP endpoints (example).

```
POST /predict            {input...}      ->
routed to v1 or v2 by local weights (current COMMITTED version)

GET  /routing/state      -> {version, weights, model_ids, status}

GET  /health/snapshot    -> {p95_ms, err_rate, n, window_id}
```

4.3 Message Summary

Name	Direction	Purpose / Key Fields
PrepareReq	Coord. → Part.	Propose new version; includes `RoutingState`, `version`, `txid`.
PrepareResp	Part. → Coord.	Vote `COMMIT`/`ABORT`; reason.
Decision	Coord. → Part.	Final decision `COMMIT`/`ABORT` for version.
Ack	Part. → Coord.	Acknowledge decision applied.
HealthSnap	Peer ↔ Peer	$p95$, error rate, window id, sample size n.
Heartbeat	Peer ↔ Peer	`last_committed_version`, digest.

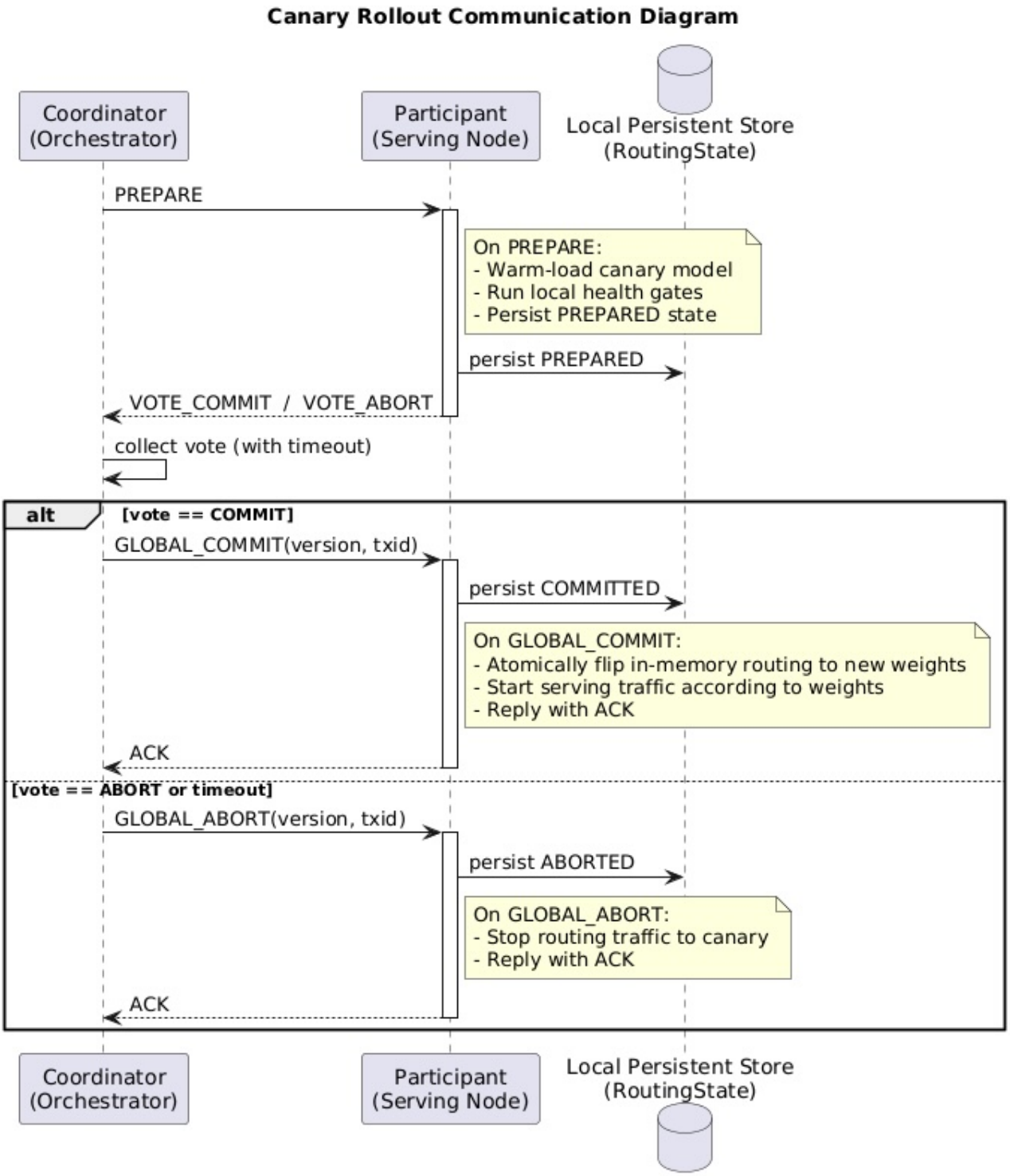

Figure 2: Canary rollout communication: coordinator issues PREPARE; participants stage the canary, run health gates, persist `PREPARED`, and vote. On unanimous commit the coordinator sends GLOBAL_COMMIT (nodes persist `COMMITTED` and atomically flip routing); otherwise GLOBAL_ABORT. All decisions are acknowledged and durably recorded in the local `RoutingState` store.