

Reconciliation Loops for Self Healing Database Operators

Sneha Gullapalli

Principal Software Engineer | Distributed Systems and Observability

Abstract

A database operator must recover from interrupted work without assuming that the previous attempt completed or failed cleanly. This paper presents a reconciliation design for managing distributed database clusters through repeatable comparisons between desired and observed state. The reconciler decomposes provisioning, topology changes, configuration updates, schema synchronization, and recovery into bounded transitions. Each transition is idempotent, records durable progress, and verifies database-level health before advancing. The design also separates declarative intent from transient operational status, uses generation numbers to reject stale work, and limits concurrent changes that could reduce availability. ClickHouse provides a concrete example because its sharding, replication, Keeper coordination, and asynchronous background behavior expose the shortcomings of one-shot automation. The proposed framework is a reference architecture rather than a report of a production benchmark.

Keywords database operator, reconciliation loop, distributed systems, idempotency, self healing, ClickHouse

1 Database Automation Needs Convergence

Traditional deployment scripts assume a linear sequence: create infrastructure, install software, write configuration, start services, and declare success. That model breaks when a process crashes after completing step four but before recording it, when an API returns a timeout after applying a change, or when another controller modifies the same resource. Retrying the entire script can duplicate work or replace healthy state. Skipping the retry can leave the cluster partly configured.

Kubernetes controllers use a different model. A controller repeatedly observes current state, compares it with the specification, and makes a limited change that moves the system toward the desired state [1]. The operator pattern extends this control loop to applications whose operational knowledge is not built into Kubernetes, including databases, upgrades, backups, and recovery [2]. For a database, convergence must cover two layers: orchestration objects such as Pods and volumes, and the database's own topology, metadata, replication, and health.

2 State Model

A database custom resource should keep desired intent in its specification and observed facts in status. The specification can define software version, shard count, replicas per shard, storage class, configuration revision, and maintenance policy. Status should report the observed generation, active topology, member health, schema revision, ongoing operation, and conditions such as Ready, Progressing, or Degraded. Mixing these concerns causes the controller to treat temporary failures as user intent or to overwrite an administrator's change with stale status.

Every accepted specification receives a generation number. A reconciliation task captures that generation before it acts and checks it again before committing status. If the user changes the specification during a rolling upgrade, work based on the old generation must stop at a safe boundary. This prevents an older task from reporting success over a newer desired state.

3 Reconciliation Pipeline

One large reconcile function is difficult to retry and difficult to reason about. A safer design evaluates ordered stages and performs at most one meaningful transition per pass. The next event, watch notification, or scheduled resync continues the operation.

Stage	Question	Safe action
Validate	Is the requested topology supported?	Reject invalid intent without changing the cluster
Discover	What resources and database members exist?	Build a versioned observed-state snapshot
Plan	What is the smallest safe difference?	Select one transition and its preconditions
Apply	Can the transition be repeated safely?	Use deterministic names and idempotency keys
Verify	Did the database reach the expected state?	Check membership, replication, and query health
Report	What should users and later loops know?	Persist conditions, progress, and retry timing

Table 1 Bounded stages in a database reconciliation pass

4 Idempotency and Partial Failure

Idempotency means that repeating an operation produces the same acceptable state, not that every API call is harmless. The operator should derive resource names from stable cluster and member identities, use create-or-patch semantics, and compare configuration hashes before writing. Database commands require stronger handling. Before issuing a schema or membership change, the controller records an operation identifier and its preconditions. On retry, it first inspects the database to determine whether the intended effect already exists. A timeout is therefore classified as an unknown outcome, not an automatic failure.

Durable phase records prevent ambiguous recovery. A topology change might move through Planned, MemberCreated, Joined, DataSynchronized, TrafficEnabled, and Completed. The operator advances only after verifying the previous effect. If it restarts in DataSynchronized, it does not recreate the member; it checks replication state and continues. Compensation should be used selectively. Deleting a new replica may be safe before it receives traffic, while rolling back a schema mutation after clients depend on it may be riskier than completing the forward change.

5 ClickHouse Example

A ClickHouse operator illustrates why infrastructure readiness is insufficient. A Pod can be Running while its replica is absent from the expected topology, is lagging, or cannot communicate with ClickHouse Keeper. Keeper provides coordination through Raft, while replicated table state can converge asynchronously across nodes [3, 4]. The reconciler must therefore combine Kubernetes observations with database-level checks.

During scale-out, the operator creates one member, waits for service discovery and Keeper connectivity, confirms that replicated metadata is present, checks synchronization, and only then exposes the member to query traffic. During a rolling upgrade, it verifies that remaining replicas can serve the workload before taking one member out of service. ClickHouse's official operator similarly uses a StatefulSet per replica to retain fine-grained control over version and configuration [5]. These checks should be policy-driven because acceptable lag and quorum differ across clusters.

Schema reconciliation needs its own ownership rule. The operator may manage a declared schema revision, observe schema only, or delegate all DDL to applications. It should not combine these modes implicitly. ClickHouse can propagate metadata through replicated database mechanisms, but asynchronous application means the operator still needs a completion condition and timeout [6]. A failed DDL must remain visible as an operation condition rather than being retried indefinitely.

6 Scheduling and Safety

Watch events provide fast reaction, but periodic resynchronization is still necessary because database state can change outside the Kubernetes API. Cached reads may lag behind writes; reconciliation must tolerate an extra invocation and must not assume immediate read-after-write consistency [7]. Retries should use exponential backoff with jitter for

transient faults and a slower scheduled requeue for long-running synchronization. Invalid specifications and failed safety preconditions should not trigger tight retry loops.

Concurrency is controlled per cluster and, where possible, per operational domain. Backup scheduling can proceed while a status refresh runs, but a topology reduction should not overlap a version upgrade. A lease can provide active-passive controller leadership, yet business-level operation records are still required because leadership can change after an external effect and before status is written. Destructive steps require explicit policy, a validated backup or recovery path, and a finalizer that prevents resource deletion until the operator has handled external state.

7 Observability and Evaluation

The operator should emit a trace for each reconciliation pass, structured events for state transitions, and metrics for queue depth, reconcile duration, retries, failed preconditions, and time spent in each operation phase. Logs should include cluster identity, generation, operation identifier, and transition name. High-cardinality identifiers belong in traces and logs rather than unbounded metric labels.

Evaluation should inject failures between every pair of durable phases: process termination, API timeout, stale cache, Keeper loss, node restart, and specification change. A successful test proves eventual convergence without duplicate members, unsafe availability loss, or false Ready status. Other useful measures include recovery time, number of external calls per convergence, retry amplification, and the percentage of failures that require human intervention. These tests assess controller correctness; they should not be replaced by a demonstration in which every dependency behaves normally.

8 Conclusion

A self-healing database operator is reliable when interrupted work can be resumed from observed facts. Small transitions, idempotent effects, generation checks, database-level verification, and durable progress records turn retries from a hazard into the normal recovery mechanism. For ClickHouse and similar distributed databases, the operator must reconcile both infrastructure and internal database state. The practical starting point is narrow: automate provisioning and one-member-at-a-time recovery, test failure at every phase boundary, and expand authority only after convergence behavior is measurable.

Declaration of AI Assisted Writing

OpenAI Codex [8] was used during preparation of this manuscript to assist with development of the initial draft, organization of the technical discussion, language editing, reference discovery, and document formatting. The author supplied the subject matter, professional context, technical direction, and review criteria. The author subsequently reviewed and revised the manuscript, verified the cited sources and technical statements, and accepts full responsibility for the accuracy, originality, interpretation, and final content of the work.

References

- [1] Kubernetes Authors, "Controllers," Kubernetes Documentation. [Online]. Available: <https://kubernetes.io/docs/concepts/architecture/controller/>.
- [2] Kubernetes Authors, "Operator Pattern," Kubernetes Documentation. [Online]. Available: <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>.
- [3] ClickHouse, Inc., "ClickHouse Keeper," ClickHouse. [Online]. Available: <https://clickhouse.com/clickhouse/keeper>.
- [4] A. Schulze, "ClickHouse: Lightning Fast Analytics for Everyone," Proc. VLDB Endowment, 2024. [Online]. Available: https://clickhouse.com/uploads/Click_House_Schulze_1_c12ecfaed4.pdf.
- [5] ClickHouse, Inc., "Introducing the Official ClickHouse Kubernetes Operator," Jan. 29, 2026. [Online]. Available: <https://clickhouse.com/blog/clickhouse-kubernetes-operator>.
- [6] ClickHouse, Inc., "No More Disks: The Architecture Behind Stateless Compute in ClickHouse Cloud," July 2025. [Online]. Available: <https://clickhouse.com/blog/clickhouse-cloud-stateless-compute>.
- [7] Kubernetes Authors, "How the controller-runtime Cache Actually Works," Kubernetes Blog, July 29, 2026. [Online]. Available: <https://kubernetes.io/blog/2026/07/29/controller-runtime-cache-explained/>.
- [8] OpenAI, "Codex," OpenAI Developers. [Online]. Available: <https://developers.openai.com/learn/codex>.