# Towards Practical Self-Healing Distributed Databases

Joseph Lynch
*Cloud Data Engineering*
*Netflix, Inc.*
Los Gatos, USA
josephl@netflix.com

Dinesh Ashok Joshi
*Apache Cassandra*
*Apache Software Foundation*
San Jose, USA
djoshi@apache.org

***Abstract*—As distributed databases expand in popularity, there is ever-growing research into new database architectures that are designed from the start with built-in self-tuning and self-healing features. In real world deployments, however, migration to these entirely new systems is impractical and the challenge is to keep massive fleets of existing databases available under constant software and hardware change. Apache Cassandra is one such existing database that helped to popularize "scale-out" distributed databases and it runs some of the largest existing deployments of any open-source distributed database.**

**In this paper, we demonstrate the techniques needed to transform the typical, highly manual, Apache Cassandra deployment into a self-healing system. We start by composing specialized agents together to surface the needed signals for a self-healing deployment and to execute local actions. Then we show how to combine the signals from the agents into the cluster level control-planes required to safely iterate and evolve existing deployments without compromising database availability. Finally, we show how to create simulated models of the database's behavior, allowing rapid iteration with minimal risk. With these systems in place, it is possible to create a truly self-healing database system within existing large-scale Apache Cassandra deployments.**

***Index Terms*—Distributed Databases, Control Planes, Apache Cassandra, Databases**

## I. Introduction

Advances in data storage have proven that cheap commodity hardware is a viable platform for large scale databases, with significant improvements shown in data models, distribution mechanisms, consensus protocols, and storage formats. Despite these improvements, operating existing database technologies remains a high-touch activity. Operators are forced to either invest heavily in techniques for coping with change, or to resign themselves to operating a system which imposes stasis rather than enabling dynamism.

Meanwhile, a new class of databases offers significantly increased automation "out of the box". These self-tuning and self-healing databases promise significantly reduced operator burden, with the expectation being that the database software itself handles ever more aspects of its management, raising the level of abstraction visible to the database operator even further. We explore the methods and techniques used to produce such a self-healing database, and evaluate their application to an existing technology: Apache Cassandra. Our hope is that existing deployments can achieve significant improvements in operability without requiring them to change the aspects of the database that work for them. By doing so, operators can continue to reap the benefits which originally attracted them to Apache Cassandra, while reducing maintenance burden and enabling even greater scale.

Apache Cassandra is a widely used open source distributed database that has gained popularity for its flexible data model, tunable consistency guarantees, and linear scalability through use of partitioning. When deployed at large scale, however, problems arise which the open-source database does not address on its own. Faults are common in both software and hardware. Meanwhile, the system must deal with changes in its own requirements: software upgrades, changing traffic patterns, data load, etc. Most operators write automated scripts to monitor and deal with such issues. However, over time it becomes extremely complex to manage large deployments without support for self-healing built into the database. Apache Cassandra wasn't designed with self-healing in mind and therefore it is interesting to explore a minimal design to build self-healing properties into a mature distributed database like Apache Cassandra.

This paper first introduces the challenge of operating a large scale Cassandra deployment. Next we describe the building blocks of a self-healing system. Finally, we discuss how to safely experiment on production deployments.

## II. Goal of Self-Healing Distributed Databases

A self-healing distributed database does not imply that it handles all types of failures and magically tunes itself to the most optimal configuration. Instead, the main goal of a self-healing database is to constantly move towards desired goals limited to self-preservation. When such a database takes action to "heal" itself, it is critical to have sufficient transparency for the operator to understand the rationale behind the database's actions [1].

To be useful in practice, the database's self-healing capabilities must have thoughtful limitations put in place, both to help human operators understand what is happening and to limit business risk. Theoretically, it may be possible to train a black-box machine learning model such as a deep neural network or a genetic algorithm to seek optimal database configuration, but we believe the field lacks the ability to express complex

constraints like durability or availability risk in the required objective function(s) [2]. In particular, black-box cost/objective function optimization techniques often prioritize properties that are easy to measure, such as performance, over those that are truly important but difficult to quantify such as durability or availability risk. Furthermore, in industry we often witness intelligent systems enter oscillating states, and if the system cannot explain itself, the human operator is likely to disable it entirely.

## III. Operational Challenges of Large Scale Apache Cassandra

Apache Cassandra's design [3] is derived from Amazon's Dynamo [4] and Google's BigTable [5] distributed database systems. The basic architecture is a collection of servers that form a Chord-style distributed hash table (DHT) [6] and use a Log Structured Merge (LSM) tree storage engine [7]. It is a quorum based eventually consistent system with client-tunable consistency on write and read operations.

### A. Deploying and Managing Cassandra

Deploying and running a distributed database such as Apache Cassandra can be challenging. In addition to having to manage the software on a cluster, hardware is constantly degrading and failing. In order to ensure smooth operations, operators monitor Cassandra's metrics and perform operations using control interfaces that the database exposes. Often, operators aggregate metrics in an external system in order to monitor trends over time. Trends, such as elevated read or write latency, can help detect issues before they manifest as user impacting problems. Today, operators are notified when trends breach acceptable thresholds. In a typical Cassandra setup operators manually intervene to solve issues such as hardware failure, hardware degredation, and software or configuration bugs. A self-healing database would ideally solve some subset of these issues for the operator.

### B. Common Control Interfaces

Apache Cassandra enables tuning via several interfaces. The basic configuration is stored in a YAML file which is loaded only at start-up time. For changes which must be applied online, Cassandra offers a Java Management Extension (JMX) operator interface which can modify a running database process. Finally, Cassandra has `system` tables which modify configuration through Cassandra's native CQL protocol.

### C. Administrative overhead during scale out

Maintenance tasks can be divided into two categories: those which can be performed on a node while it still services queries, and those which require downtime of at least one node in the cluster. The former category includes many types of tuning and reconfiguration, as well as modifications to hardware systems which are redundant (e.g. power supply units and shared block storage). The second category encompasses most software upgrades and modifications of non-redundant hardware systems (e.g. commodity CPUs and motherboards). A useful metric to consider is how long the database can continue operating without human intervention.

## IV. Self-Healing Architecture

The fundamental components of a self-healing distributed database deployment are *Specialized Agents* that can sense and react to events. These agents cooperate to keep the database functioning through the *Cluster Manager* which aggregates state and mediates state transitions for the agents. The system draws inspiration from the human nervous system which not only senses but also can react locally or globally based on sensory inputs.

Crucially, every component of the system is "goal-driven": they have some conception of the desired goals for the cluster, and are constantly working to converge towards that goal state. The system starts with a declarative description of what the human operator wants the database to accomplish. These descriptions are written as hierarchical configuration documents parameterized by environment that describes all aspects of the database's deployment for that environment. For example, an Apache Cassandra goals document might include fields such as:

```
goals(cluster_name, environment) = {
    software = {
        os_version = "bionic_20191023",
        db_version = "3.0.19.7",
        db_config  = {...}, ...
    },
    hardware = {
        node_count_range = [48, 96],
        node_type = "m5d.2xlarge", ...
    },
    service_level_objectives = {
        read_latency_p99_ms = 10,
        1m_availability = 0.999,
        max_cost_usd = 100000, ...
    }
}
```

This document should completely describe the operator's desired goals to the Cluster Manager, which can then delegate to specialized agents, each with a limited area of responsibility. Configuration is hierarchical so that operators can express goals at a granular level and then the configurations "merge up" the environment hierarchy. For instance, an operator could set the `hardware.node_type` goal for only one availability zone within a region, or could specify an override `db_version` for an entire cluster in `staging`.

Service Level Objectives (SLOs) are included in the goal document so that the Cluster Manager can monitor the database system's achievement of business-critical metrics [8] such as availability of the system for consistent read queries or the percentage of write queries that are exceeding the write latency target. One can also include less direct objectives such as expected time between failures so that the agents

can appropriately rate-limit their maintenance actions based on failure models [9].

In addition to goals, the system associates *context* about the data stored within each cluster. Unlike goals, context primarily impacts user access, monitoring, and incident response flows rather than day to day maintenance activities.

```
context(cluster_name, environment) = {
    owners = ["owners@company.com", ...]
    users = ["user1", "app1", ...]
    tags = {
      sensitive = true,
      tier = 1,
    }
}
```

A common purpose of this context is to help inform monitoring systems to determine which human engineers care about a database and why. For example, at Netflix, our systems might page owners of a tier zero cluster when SLOs are breached instead of notifying via email. Another common use of context is to allow automated access control for the appropriate engineers to the appropriate clusters. As an example, databases that contain sensitive data might default to restrictive access control or provide for "taint" tracking in downstream data pipelines. This capability allows incident response involving data exposure to understand where sensitive data originated and how limited the impact is.

The self-healing database architecture seen in Fig. 1 constantly drives towards the desired goals subject to the constraints imposed by the context. For each goal we surface a measurement of the state of the system from local agents, and when the local agent detects that the node state does not meet the desired goal state, that specialized agent takes action to move towards the desired state. When coordination with other agents is necessary, state transitions are communicated through the Cluster Manager. Note that a self-healing system cannot synchronously issue commands to the agents as we cannot guarantee they will be delivered, instead the Cluster Manager facilitates communication through state transitions stored in a state database. The specialized agents are responsible to asynchronously observe these state changes, act, and update any needed state. Even if the Cluster Manager is unavailable, the agents can still make progress on tasks that do not require coordination. This is conceptually similar to the Propagator programming model where autonomous machines (agents) communicate through shared state to create an intelligent system [10] [11].

## V. Supervisor Agents

The building blocks of self-healing database systems are a flexible set of specialized supervisor agents that run on every node. These agents collect metrics, emit them, and execute modifications to their respective nodes along with reporting feedback about the state of the world to the Cluster Manager. Agents may not always execute modifications, they might only collect and emit metrics. The composition of multiple agents, each with its own specialization, yields a self-healing system that is simultaneously powerful *and understandable*. Importantly, the set of agents is not limited; as new failure modes are discovered existing agents may either take on the scope of the failure or a new agent is created.

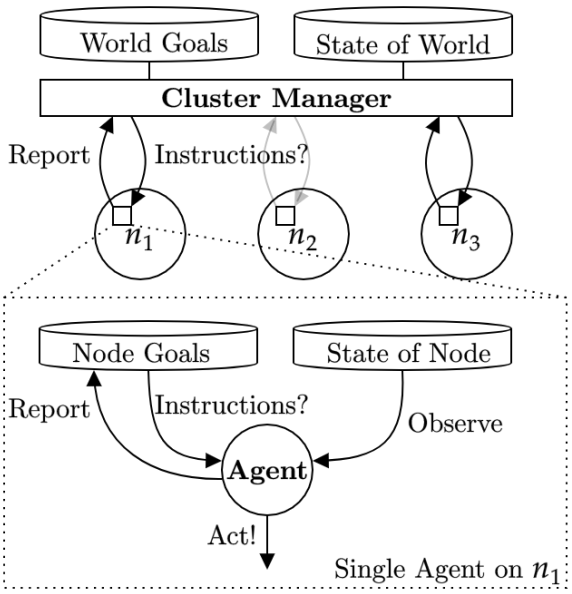


Fig. 1. Goal-Driven Self-Healing Distributed Database Architecture

### A. Key Metrics for Service Level Objectives

Like many databases, Apache Cassandra emits a multitude of metrics that help operators understand what is happening in the system. Operators can attach dedicated Java Virtual Machine (JVM) agents to Cassandra and additional specialized agents to the system which collect these key metrics and continuously emit them. These metrics allow operators to establish SLOs over business metrics such as throughput and latency.

For the purposes of self-healing it is important to separate metrics that are *contextual* from critical *call to action* metrics emitted by agents. Contextual metrics are useful to capture and provide as context during a call to action, such as CPU utilization. Call to action metrics measure SLO impacting business metrics and might include metrics such as the system's error rate, average latency, and $99^{th}$ percentile latency. Self-healing databases only act due to changes in call to action metrics, but whenever it takes an action based on a call to action metrics it considers contextual metrics and summarizes the context to present to operators via dashboards or reports. In practice, operators often reject intelligent database systems that do not provide contextual summaries explaining their actions, so a self-healing database must be able to explain itself.

### B. Hardware Supervisor Agents

Apache Cassandra commonly handles individual queries in under one millisecond. Given this stringent standard, even

small hiccups in a server's hardware performance can produce SLO breaking outages. In particular, due to Cassandra's heavy usage of memory-mapped I/O, it is extremely vulnerable to latency induced by drive slowness.

To compensate, a self-healing Cassandra deployment must constantly interrogate its hardware to ensure key system properties are holding true. When these properties no longer hold true, agents work with the Cluster Manager to remove problematic hardware from service. For example, a new Cassandra server can run hardware diagnostics on first boot to reduce latent failures:

**Example Startup Hardware Supervisor Agents**

- A disk burn-in check uses `fio` [12] to test drives.
- A network probe checks network connectivity.

If the provisioned hardware cannot meet the stringent disk latency SLO or the network SLO it is immediately rejected from the cluster without joining. At Netflix, these two simple startup checks reduced the number of latent hardware failures by ejecting degraded hardware from clusters early. Netflix launches thousands of cloud database instances every week during backup restoration drills, and around 1% of them fail to meet SLOs within that week which requires engineers to debug and remove them from service manually. After preflight checks were introduced this number was reduced to near zero.

Once a server is running, it must continue to interrogate the hardware, continuously assessing if the hardware remains within specification:

**Example Continuous Hardware Supervisor Agents**

- Failed drive: Perform a small I/O to the data mount.
- Slow drive: Inspect the OS block device service time excluding queuing time [13]. This differentiates a slow drive (requires replacement) from an overloaded one (requires scale up).
- Processor overload: Inspect the OS CPU scheduler delays [14] to measure how long threads queue waiting to run on a core. This gives an accurate signal for if adding cores would alleviate load.
- Network partitions: Send periodic network heartbeats.

These specialized agents provide the domain context the Cluster Manager needs to understand *why* latency is happening. Rules-based agents for these small subsets of hardware failures are sufficient to self-heal most failures; no complex machine learning model needed. In practice, these kinds of simple checks detect and recycle thousands of cloud instances at Netflix every year that otherwise would require manual human intervention.

### C. Software Supervisor Agents

Just as hardware agents can detect certain common hardware faults, specialized software agents can help detect anomalous activity in the Cassandra database itself and act to remedy the fault. Two simple but useful self-healing agents are:

**Example Software Supervisor Agents**

- A generic process supervisor such as `systemd`. This detects failed processes and restarts them to preserve availability (emitting logs for context).
- A JVM supervisor such as `jvmquake` [15] which detects unstable JVMs and kills them (emitting a core dump for context). This action plus the process supervisor restores availability.

These agents are simple, but they self-heal a large number of real-world faults. Furthermore, the quick reaction time of localized agents minimizes Mean Time To Recover (MTTR) when faults do occur to mere seconds. At Netflix, the JVM supervisor has been particularly impactful, reducing the duration of "query of death" [1] outages from hours to mere minutes. These kinds of simple agents, however, operate with safety guards in place. In particular, crash-recover loops are rate-limited and when loops are detected, explanatory reports are issued to operators to aid in debugging.

Another class of software supervisor agents involve supervising self-healing tasks of the Cassandra database cluster.

**Example Cassandra Supervisor Agents**

- An incremental `backup` agent that exploits LSM storage engine to provide point-in-time snapshots.
- An incremental `repair` agent which guarantees eventual consistency.
- A `replacement` agent that detects permanently failed servers and restores data to new cluster members from either a durable backup or other replicas.

Examples of software systems that implement these kinds of Cassandra specific supervisor agents are:

- Netflix's `Priam` [16] co-process which configures and manages Apache Cassandra to run in the AWS cloud.
- The `Reaper` [17] repair manager from Spotify and The Last Pickle which handles scheduling repair operations on Cassandra clusters.
- Soon, the Apache Cassandra Sidecar [18] which will build these supervisory functions directly into an official sidecar process for Apache Cassandra.

These software supervisors act as a local machine agent that both perform these required supervisory functions as well as

[1] "Queries of death" are when a particularly expensive query is issued against the cluster that causes it to die a slow death, in Apache Cassandra and Elasticsearch most frequently due to being asked to load the entire dataset into heap memory

act as a local agent which the Cluster Manager can instruct to perform actions.

A key feature of agents which we have found useful in practice is for them to be engineered to be crash-tolerant by reacting to serious faults in a crash-only fashion [19] [20]. Crash-tolerant agents make incremental progress toward their goals under the assumption they may have to crash and resume at any time. Crash-only agents choose to handle most serious faults by explicitly crashing rather than attempting to recover. We find that crash-only agents exercise crash-tolerance more frequently and force engineers to externalize state which increases the likelihood the agents can handle true faults [21].

Netflix's initial implementations of `repair` and `backup` agents were not crash-only: they lost their entire progress after encountering a fault and being unable to recover from the fault. We found this untenable in practice because faults proved so common that these agents were not able to complete their tasks and would have to restart from the beginning of the task. In practice, this meant that certain large-scale, typically petabyte-scale, clusters failed weekly to reliably complete their backups or data repair tasks. After switching to crash-only implementations which constantly check-pointed progress state so that the agents could always resume, all Cassandra clusters were able to successfully backup and repair regardless of scale.

Choosing a goal-oriented and crash-only architecture for these local software agents in practice makes the distributed database significantly better at maintaining a desired goal state without significant additional complexity. For example, at Netflix, implementing these semi-intelligent purpose-built agents reduced operator actions by an order of magnitude.

## VI. Goal-Driven Cluster Manager

Armed with contextual metadata reported by the specialized agents, the *Cluster Manager* is responsible for evaluating the state of the cluster and taking actions to achieve or maintain the operator's desired goal. It also presents operators with a linear history of every state transition and action taken by agents. This linear history is critical for the system to be understandable and, in future work, for anomaly detection.

### A. Aggregation and Storage of State

One key role of the Cluster Manager is to aggregate reports from agents across the fleet into a useful cluster and fleet level summary. This gives agents a location in which to checkpoint their state. The Cluster Manager also provides a framework for agents to save their positions in distributed state machines, which can be used, for example, to implement locking. This allows agents to each individually take availability-affecting actions, while enforcing semantics that keep the database as a whole healthy.

Internally many distributed databases have their own version of this control-plane state, for example cluster membership or schema changes might use strongly consistent operations based on Raft [22]. The Cluster Manager design intentionally does not use database specific control-plane storage so as to minimally couple to the underlying databases the agents manage.

### B. Taking Action to Meet Desired Goals

The Cluster Manager needs a relatively small number of actions to solve a wide range of database faults. The majority of complexity is delegated to the specialized agents that follow purpose-built state machines. As the specialized agents contain most of the intelligence, high level actions have narrow scope:

**Cluster Manager Actions**

1) Write a new agent goal.
2) Record a state transition from an agent.
3) Add or remove servers to the cluster.
4) Alert humans.

Assuming a cloud computing environment composed of virtual servers running a composition of containers (pods) per instance of the database, the Agent Actions are limited to:

**Specialized Agent Actions (in order of growing risk)**

1) Actuate a live (JMX) control interface (online)
2) Upgrade a single non-database container (online)
3) Re-image the server to a new machine image
4) Transfer data to another server and exit.
5) Exit the cluster without data transfer

Online actions (Agent Actions #1-2) do not affect database availability, and so agents can generally perform them without coordinating through the Cluster Manager. Agents must coordinate through the Cluster Manager's state functionality (Cluster Manager Action #2) for any action which stops the local instance of the database (Agent Actions #3-5). Cassandra's data distribution algorithm lends itself easily to a state machine: at any given time, only nodes in a single Cassandra rack can safely perform offline maintenance.

The agents are programmed to understand which actions can be used to meet their goals, and always select the lowest-risk candidate action from the above list. For instance, the configuration management agent understands that a subset of properties can be actuated via the live control interface and will choose to use it instead of re-imaging a server with new configuration if possible. Another example is both the repair and backup agents choose to solely interact with online interfaces as they operate continuously.

Luckily these five actions are simple to choose between, it is always optimal to choose the least risky option that allows an agent to meet the goal. Specifically, actuating a live configuration knob is always safer than changing software because it is fast and usually safe to revert. Changing a non-database container is always safer than re-imaging the machine because every time the database restarts there is risk to availability [9]. Re-imaging a server in place is always safer

than migrating hardware due to the propensity for hardware to fail young [23]. The highest risk action an agent can take to meet a goal is to replace the underlying hardware as it not only involves new servers which may rapidly fail, but it also typically involves moving data between drives which provides an opportunity for data corruption.

As a concrete example, to perform a database software upgrade of a particular cluster in production, the operator writes a new desired goal to the state database:

```
D = goals(
  cluster_name = "cass_cluster",
  environment = {
    deployment="prod"
  }
)
D.software.db_version = "3.0.19.8"
```

The distributed `replacement` agent notices that the desired database version differs from its local version. The `replacement` agent understands that this requires restarting the database process. This in-turn means that it will take Agent Action 3 and re-image the server. In fact, any action that requires a restart of the database process can be simplified to re-imaging the entire database server in-place, because modern imaging techniques take roughly the same amount of time (minutes) as restarting the database process and allows atomic transitions from one tested state to another [24] [25]. In a containerized environment this is the equivalent of deploying a new pod or container of software. A common alternative to re-imaging is configuration management, but that introduces the risk of configuration drift and introduces numerous dangerous mutability seams into every database server, rather than having a single atomic mutation from one known good server image to another known good server image. Note that for maintenance that *does not* require the database process to restart it is often preferable to take Agent Action 2 instead as that is lower risk.

Instead of teaching the Cluster Manager how to perform a safe upgrade of an Apache Cassandra database, the self-healing architecture delegates that complexity to the agent which understands the domain specific guarantees and requirements. The Cluster Manager merely indicates the new desired goals.

Another role of the Cluster Manager is to react to unexpected situations. The on-instance agent may react to hardware faults by signaling to the Cluster Manager it is terminating itself (Agent Action 5). Alternatively, a network fault or cloud provider action may force-retire a node. In both cases, the Cluster Manager must launch a new instance and provide relevant context to the `replacement` agent so that the new node can re-constitute state either from backups or from neighboring replicas.

## VII. Safely Experimenting on Production

Even with a fully self-healing database deployment of Apache Cassandra, it is unwise to rely on the self-healing production system for finding optimal database settings due to the potential for availability and durability loss. Instead, in industry, a common technique is to simulate production as completely as possible.

The specialized agents described earlier, especially the `backup` agent, make simulating production straightforward. For example, out of many clusters, there are probably a small number that are truly business critical. In such a case a very safe experiment can be run:

1) Copy the cluster's goals document from production to a testing environment. The self-healing system restores from backup to an equivalent configuration.
2) Capture live query distribution from production using audit logs or tracers [26] [27].
3) Apply continuous load that matches the production query distribution [28].
4) While the load test is running, a `chaos` agent kills machines and processes to simulate failure conditions.

Operators typically choose to let these experiments run for days or weeks as there is no human effort involved and to ensure that the proposed change will not adversely affect durability, availability, or performance of the system. It is vital to have this long testing period because many faults do not immediately surface.

Once the isolated experiment is complete, a human operator views the contextual report issued by the self-healing Cluster Manager and makes the decision to "promote" the configuration to a production environment. To minimize risk the Cluster Manager incrementally applies the desired goal to low priority clusters and then to critical clusters. To apply the desired goal to a cluster the Cluster Manager first changes the goals document for a single machine in a single rack, then to all the machines in a rack, and finally to all machines in all racks. This gradual roll-out further reduces risk.

## VIII. Conclusions and Future Work

We have described a concrete architecture for building self-healing distributed databases, drawing on real-world experience with Apache Cassandra. Applying domain expertise, a relatively small number of rules-based agents can be composed together to form a self-healing system atop existing database software. Advanced central planning is not needed beyond the limited Cluster Manager, which stores desired goal state and mediates state transitions via the specialized agents. In practice, we have observed these goal-driven agents reduce operational incidents both in frequency and duration by an order of magnitude while simultaneously allowing more rapid scaling of hardware and innovation in software. In the future we envision using the state transition history to predict failures and surface anomalies.

We believe the hierarchical goal and agent architecture generalizes to any stateful system, with the key qualifier that database or language specific agents are only composed when they make sense. Indeed at Netflix we use variations of the same architecture for most of our business-critical real-time distributed databases such as Apache Cassandra, Elasticsearch, EVCache (distributed Memcached) and CockroachDB, but we

only use the `jvmquake` JVM supervisor agent for the first two as the other systems are not written in Java.

Finally, we show how to safely experiment on these database systems without resorting to hard to understand machine learning models. This cautious approach is taken in industry because business continuity depends on these systems functioning at close to 100% reliability. No database operator wants to explain, "the billing database was corrupted because a black-box model turned on a rarely used database flag which improved performance but caused silent data corruption in the entire dataset". This is one of the many reasons we advocate for understandable self-healing databases rather than fully self-tuning ones.

## ACKNOWLEDGMENT

We would like to thank the numerous engineers on our teams for contributing in many ways to the success of this architecture. At Netflix we would like to specifically acknowledge Josh Snyder for providing significant feedback on the manuscript in addition to implementing many of the systems this paper is based on as well as Prudhviraj Karumanchi, Vinay Chella, Arun Agrawal, Itay Tenne and other team members that contributed to the systems described herein.

## REFERENCES

[1] D. Gunning, "Explainable artificial intelligence (xai)," *Defense Advanced Research Projects Agency (DARPA), nd Web*, vol. 2, 2017.

[2] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang, "Self-driving database management systems," in *CIDR 2017, Conference on Innovative Data Systems Research*, 2017. [Online]. Available: https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, p. 35–40, Apr. 2010. [Online]. Available: https://doi.org/10.1145/1773912.1773922

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. USA: USENIX Association, 2006, p. 15.

[6] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan, "Building peer-to-peer systems with chord, a distributed lookup service," in *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001, pp. 81–86.

[7] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, p. 351–385, Jun. 1996. [Online]. Available: https://doi.org/10.1007/s002360050048

[8] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*, 1st ed. O'Reilly Media, Inc., 2016.

[9] J. Snyder and J. Lynch, "Cassandra availability with virtual nodes," https://jolynch.github.io/pdf/cassandra-availability-virtual.pdf, 2018, accessed: 2020-02-22.

[10] A. Radul and G. J. Sussman, "The art of the propagator," in *Proceedings of the 2009 international lisp conference*, 2009, pp. 1–10.

[11] A. Radul, "Propagation networks: A flexible and expressive substrate for computation," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[12] "Flexible i/o tester," https://fio.readthedocs.io/en/latest/fio_doc.html, accessed: 2020-02-19.

[13] "Linux i/o statistics," https://www.kernel.org/doc/Documentation/iostats.txt, accessed: 2020-02-19.

[14] "Linux cpu scheduler statistics," https://www.kernel.org/doc/html/latest/scheduler/sched-stats.html, accessed: 2020-02-19.

[15] J. Lynch and J. Snyder, "Garbage collecting unhealthy jvms, a proactive approach," https://medium.com/@NetflixTechBlog/introducing-jvmquake-ec944c60ba70, accessed: 2020-02-19.

[16] Netflix, "Netflix priam cassandra sidecar," https://github.com/Netflix/Priam, accessed: 2020-02-19.

[17] T. L. P. Spotify, "Reaper cassandra repair manager," https://github.com/thelastpickle/cassandra-reaper, accessed: 2020-02-19.

[18] T. A. S. Foundation, "Apache cassandra sidecar," https://github.com/apache/cassandra-sidecar, accessed: 2020-02-19.

[19] G. Candea and A. Fox, "Crash-only software," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS'03. USA: USENIX Association, 2003, p. 12.

[20] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, Mikroelektronik och informationsteknik, 2003.

[21] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[22] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[23] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Trans. Dependable Secur. Comput.*, vol. 7, no. 4, p. 337–351, Oct. 2010. [Online]. Available: https://doi.org/10.1109/TDSC.2009.4

[24] J. Snyder, "Rebooting datastores into the future," https://medium.com/@NetflixTechBlog/datastore-flash-upgrades-187f1e4ef859, accessed: 2020-02-19.

[25] Netflix, "S3 flash bootloader," https://github.com/Netflix-Skunkworks/s3-flash-bootloader/, accessed: 2020-02-19.

[26] A. C. Community, "Audit logging in apache cassandra 4.0," http://cassandra.apache.org/blog/2018/10/29/audit_logging_cassandra.html, 2018, accessed: 2020-02-21.

[27] J. Lynch, "Cqltrace: A dynamic tracer for viewing cql traffic in real time," https://github.com/jolynch/cqltrace, accessed: 2020-02-21.

[28] I. Papapanagiotou and V. Chella, "Ndbench: Benchmarking microservices at scale," *arXiv preprint arXiv:1807.10792*, 2018.