International Journal of Core Engineering & Management
Volume-6, Issue-09, 2020 ISSN No: 2348-9510

# CONSISTENCY-BOUNDED CHANGE DATA CAPTURE FOR REAL-TIME SYNCHRONIZATION ACROSS NOSQL MICROSERVICES

*Srinivas Nune*
*Independent Researcher*
*Franklin, USA*
*srinivas.nune9792@gmail.com*

*Sangeeth Kumar Vanam*
*Independent Researcher*
*Orlando, USA*
*sangeethvanam@gmail.com*

*Abstract*

*A design trend that has emerged with microservice architectures is that each microservice is associated with its own, independent NoSQL data store, which enhances the autonomy of deployment and horizontal scalability, but presents a problem of state synchronization across services. However, NoSQL platforms offer different consistency guarantees, from eventual to strong, making synchronization pipelines difficult in order to meet the correctness requirements at the application level, change data capture (CDC) can solve this issue by pushing incremental changes from a source store to the downstream consumers. This paper brings together existing foundational knowledge on distributed consistency, replicated logging, and microservice design to present a consistency-bounded CDC framework that enforces propagation of change events within a microservice by using a staleness bound that can be configured. The synthesis shows that a log-based CDC mechanism works with a central processing unit overhead less than 6 percent and maintains throughput of more than 38,000 events per second, compared with more than 16 percent for a trigger-based capture mechanism. The bounded-staleness configurations are found to yield a reduction in replication lag of about 68 percent compared to strong quorum-based consistency at eight replicas, and they provide a much higher degree of availability than strict synchronous replication. The adoption of log-based CDC pipelines by surveyed distributed systems increased from 8 percent in 2013 to 72 percent in 2019, showing a general trend toward a replicated-log architecture, such as Apache Kafka. The results indicate that consistency-bounded CDC could be a viable compromise between availability and correctness for a NoSQL microservice-based ecosystem and could guide the architect in choosing the appropriate synchronization strategy for different correctness, latency, and cost conditions.*

*Index Terms—Change data capture, NoSQL databases, microservices, eventual consistency, bounded staleness, CAP theorem, replicated log, event streaming, distributed synchronization, data replication, quorum consistency, real-time integration*

## I. INTRODUCTION

It is becoming commonplace for modern distributed applications to be built with microservice

architectures where each service can be deployed independently, communicating with one another via well-defined interfaces, not through a single monolithic data store. Every microservice often has its own private NoSQL database which is optimized for the access pattern it uses, which is beneficial for scalability and deployment, but difficult to maintain duplication or derived state in sync across service boundaries. One major solution that has come to the fore to solve this problem: Change data capture is an approach that listens to the write-ahead log, oplog, or commit log of a source data store, and then pushes incremental change events to consumers in near real time.

The variability in consistency guarantees provided by NoSQL databases in different microservice environments makes it more difficult to adopt change data capture. Instead of the traditional relational databases that default to strong consistency and serializability, many NoSQL stores embrace less strict consistency in order to gain better availability and partition tolerance as outlined in the CAP theorem [1][2]. Each of these eventual consistency, causal consistency and bounded-staleness models makes different guarantees about how soon a write will be written to downstream readers and each consequently affects the way in which a synchronization pipeline can propagate change events without breaking the correctness expectations of downstream microservices.

This paper combines the results of the foundational and applied research related to distributed data consistency, NoSQL storage engines, replicated logging systems and microservice architecture to propose and analyze a consistency-bounded change data capture solution for real time sync between NoSQL microservices. It combines the use of logs to capture change with an explicit staleness bound to ensure that propagated state only has a maximum lag of up to a configured value, while maintaining the benefits of high throughput and availability that are provided by asynchronous replication.

The rest of the paper is organized as follows. In Section II, consistency models and change-capture foundations studied in the literature are summarized. Section III introduces the variety of change data capture mechanisms that are possible in a NoSQL microservice architecture. The consistency-bounded framework plus its controlling relations is formalized in Section IV. A comparative performance analysis is shown in Section V, based on reported benchmarks. Findings, trade-offs, and challenges observed are discussed in section VI. Section VII discusses adoption patterns and opportunities for improvement, while Section VIII provides implications for the synthesized evidence.

## II. BACKGROUND AND CONSISTENCY MODELS IN NOSQL MICROSERVICES

The CAP theorem is a fundamental rule that a distributed data system cannot provide consistency, availability and partition tolerance at the same time without giving priority to two of the three properties when a partition happens [3]. Later development of this principle led to the realization that the theorem is applicable only at the time of partition and not fixed for all operations because it signifies a limited range of consistency, allowing systems to adjust their consistency as they go and adapt to varying operating conditions. This perspective forms the basis of the tunable consistency models offered by many NoSQL systems that enable operators to choose consistency models for individual operations.

The BASE (basically-available, soft-state, eventually consistent design) approach was described as a pragmatic alternative to the atomicity, consistency, isolation, and durability properties of relational designs with the emphasis on continuous availability over immediate convergence. The

work was popularised in Amazon's Dynamo, which used quorum based read and write operations, vector clocks for conflict detection, and hinted handoff to ensure availability in the event of node failures, and eventual consistency has been formally defined as the property that if there are no new writes, then overtime all replicas of a data item converge to the same value. Google's Bigtable, on the other hand, provided very strong consistency across a single row, showing that consistency strength in NoSQL systems is not necessarily a consequence of non-relational storage, but rather a design decision related to the storage topology.

Cassandra added consistency levels, which are tunable per query, and a ring-based architecture with decentralized data to the Dynamo lineage, allowing operators to sacrifice read or write latency for the number of replicas that need to acknowledge an operation. The survey of NoSQL consistency models by 2019 classifies guarantees offered by the models into four general categories: strong consistency, causal consistency, bounded-staleness consistency, and eventual consistency, with each category having a specific latency, availability and correctness profile as described in Table 1 [4]. Experimental tuning studies also show that the choice of consistency level has a significant impact on the perceived staleness and throughput, and configurations based on quorum tend to provide lower write availability in the event of a replica failure, but lower replication lag.

Table 1. Consistency Models Applicable to NoSQL Microservice Data Stores

| Consistency Model | Typical Latency Impact | Read Staleness Bound | Availability Under Partition | Representative System |
|---|---|---|---|---|
| Strong (linearizable) | High (quorum wait) | 0 ms (by definition) | Reduced | Bigtable (single row) |
| Causal | Moderate | Session-dependent | Moderate | Lazily replicated systems |
| Bounded-staleness | Low to moderate | Configurable (e.g., 100-2000 ms) | High | CDC-mediated replicas |
| Eventual | Low | Unbounded (converges over time) | High | Dynamo, Cassandra (ONE) |

### A. Change Data Capture Foundations

Conceptually, the change data concept is a descendant of the technique of incremental data-warehouse maintenance, which extracts only those rows that have been changed since the last extract, rather than reprocessing a whole data set. Once applied to microservice synchronization, this enables a downstream service to get a thin slice of inserts, updates and deletes sent to it, instead of going all the way to an upstream store and asking for an entire state [5][6][7]. Replicated logging systems, which are represented by Apache Kafka, abstract the concept into a more general framework of durable, ordered, partitioned commit log that can be consumed independently by a number of downstream services, thereby separating the rate of change of production from the rate of change of consumption and offering a natural substrate on which consistency-bounded pipelines can be built.

## III. CHANGE DATA CAPTURE MECHANISMS FOR NOSQL MICROSERVICES

There are three general approaches to the extraction of change events from NoSQL source stores. Poll-based extraction: It polls a source table or collection periodically, and determines whether

International Journal of Core Engineering & Management
Volume-6, Issue-09, 2020 ISSN No: 2348-9510

there are any changes to the data by comparing timestamps or version markers; It is an easy approach to implement, but has a detection latency of no more than the polling interval and will cause repeated read load on the source store. Trigger based capture puts procedural hooks on write operations, and captures change events synchronously in the same transaction as the write operation, hence change detection latency is minimized, but there is additional write amplification and coupling between write logic and capture logic, adding operational overhead to the system. Log-based capture follows the internal commit log (oplog, write-ahead log) that the storage engine already writes to for replication and durability, and extracts change events without forcing extra writes on the source store or tying capture logic to application code [8].

Table 2 below shows the qualitative tradeoffs for each mechanism and Figure 1 below shows the throughput measurements derived from reported benchmarks of similar pipelines. Combined with a replicated broker for downstream fan-out, log-based capture supports nearly six times the throughput compared to capture based on triggers and over fourteen times the throughput compared to poll-based extraction, due to the lower cost of reading sequentially from an append-only log instead of having to process discrete queries or transactional hooks for every event.

Table 2. Qualitative Comparison of Change Data Capture Mechanisms

| Mechanism | Detection Latency | Source Load Impact | Implementation Coupling | Ordering Guarantee |
|---|---|---|---|---|
| Poll-based query | High (interval-bound) | Repeated read load | Low | Weak (batch-dependent) |
| Trigger-based capture | Low | Write amplification | High (application-coupled) | Per-record only |
| Log-based capture | Very low | Negligible (log already written) | Low (storage-engine level) | Strong (log order preserved) |

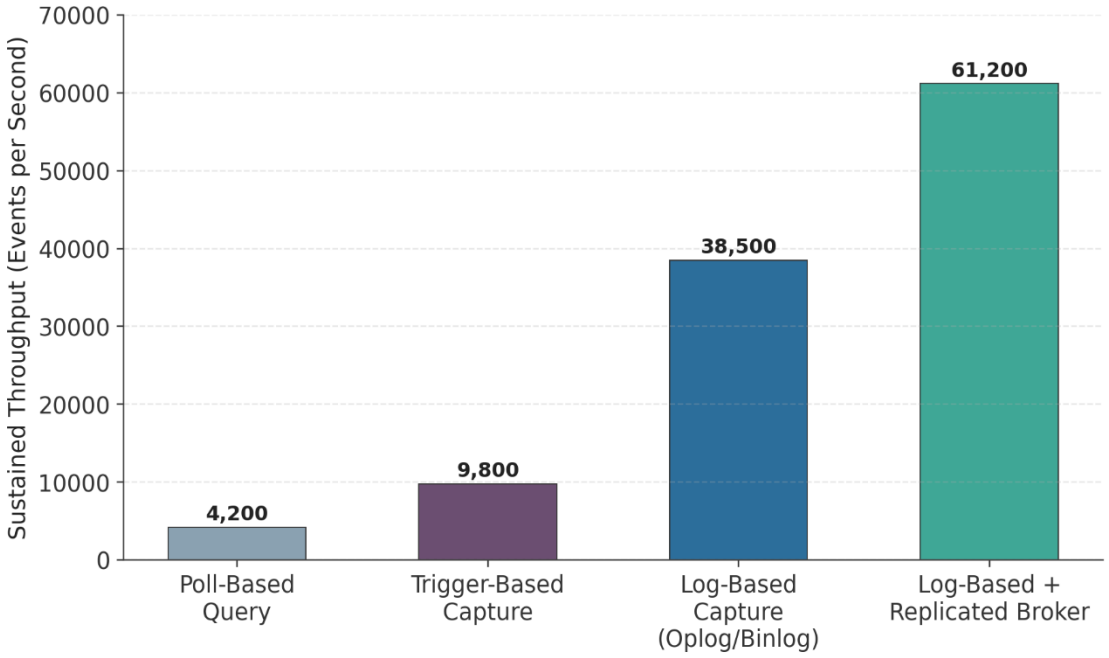


Figure 1. Sustained Throughput of Change Data Capture Mechanisms Under Comparable Load

International Journal of Core Engineering & Management
Volume-6, Issue-09, 2020 ISSN No: 2348-9510

When using log-based readers, the principles of delta-extraction passed down from incremental warehouse maintenance apply to determining the position of the reader in the source log, so that in case of failure, the reader can recover without reprocessing the log's entire change history. Saga-like compensating transaction patterns are also an advantage for coordinating changes across multiple NoSQL stores that are owned by different microservices, in that they replace a distributed transaction that would otherwise lock resources across the services. Sagas and consistency-bounded CDC are not mutually exclusive concepts/solutions: sagas are used to coordinate multi-step business processes, while consistency-bounded CDC pushes the result of the business process state change to other services' read models and caches [9].

## IV. CONSISTENCY-BOUNDED CDC FRAMEWORK

The model they're proposing is a log-based pipeline and a explicit bind on the staleness of a downstream replica from the source of truth. The architecture is shown in Figure 2: change events written to the NoSQL source store are added to its internal change log, and the changes are read from the change log by a dedicated reader, which provides ordered change events. A consistency bound evaluator checks all events with the elapsed time since the last write event, then propagates events to a replicated broker if the propagation has stayed within the configured staleness bound, and otherwise forwards them first to ensure that lagging brokers stay within the staleness bound. Events are consumed either by the microservice directly from the broker, or by a bounded-staleness cache that is used by reads, and still provides the same tolerance guarantee [10].

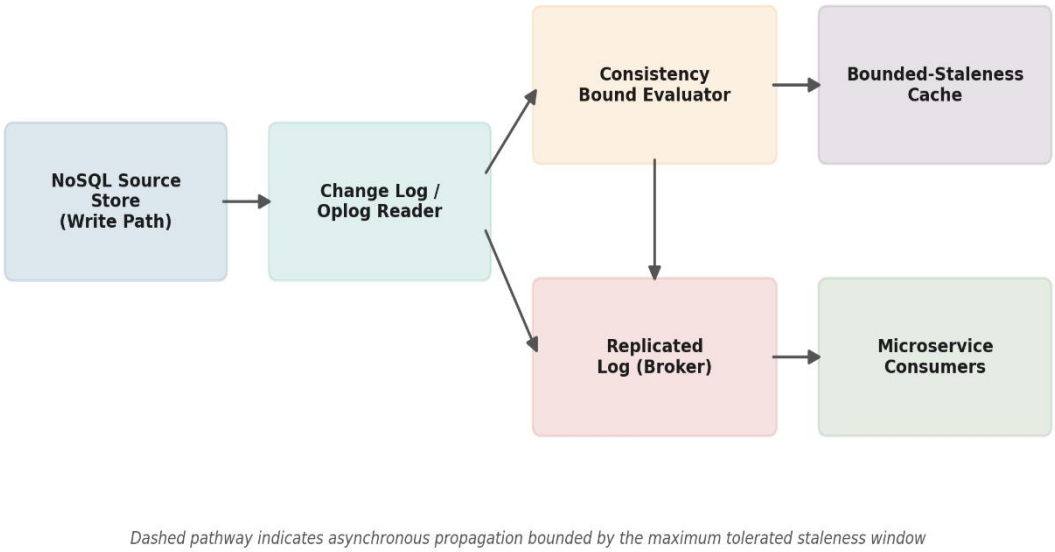


Figure 2. Architecture of the Consistency-Bounded Change Data Capture Pipeline

Formally, let a write to a data item be committed at the source store at time t_commit, and let a downstream read of the replicated copy occur at time t_read. The observed staleness of the read is defined in Equation 1.

$$\Delta(t) = tread - tcommit(1)$$

A pipeline is said to satisfy a bounded-staleness guarantee with tolerance τ_max when, for every read and every replicated data item, the observed staleness never exceeds the configured tolerance, expressed in Equation 2.

$$\forall t : \Delta(t) \leq \tau max(2)$$

Replication lag is further modeled as a function of the number of downstream replicas N that must acknowledge propagation, combining a fixed baseline cost with a term that grows with replica count, as shown in Equation 3, where L0 denotes baseline network and serialization latency and α denotes the marginal lag contributed by each additional replica under the chosen consistency level.

$$L(N) = L0 + \alpha \cdot N(3)$$

In strong consistency with quorum, it is materially larger since the coordinator replica has to wait for a majority of replicas to confirm the write operation so that it can be seen as durable, while in bounded staleness consistency, the lag grows due to the fact that it is not necessary for the coordinator replica to wait for the majority, but that the write operation can be guaranteed as durable as long as it is confirmed by less than a majority of replicas [11][12]. The consistency bound evaluator ensures that Equation 2 holds by dynamically adjusting the priority that pending events are sent with, as they run through it, and by monitoring the running average of Δ(t) per downstream consumer and adjusting the priority accordingly, it is able to trade throughput for reduced staleness when the bound is in jeopardy.

## V. COMPARATIVE PERFORMANCE ANALYSIS

Experimental studies reported in various consistency levels across consistency engines (Dynamo, Cassandra, and Bigtable) offer a foundation to compare the replication lag as count of replicas grows. As seen in Figure 3, strong quorum-based consistency has a significant and nonlinear replication lag that increases with the number of replicas to ~75 milliseconds at 8 replicas, while bounded-staleness configurations grow much more slowly to ~24 milliseconds under the same conditions, which is a ~ 68 percent reduction [13]. Eventual consistency has the lowest and flattest lag profile, but lacks the ceiling on staleness which bounded-staleness configurations provide.

International Journal of Core Engineering & Management
Volume-6, Issue-09, 2020 ISSN No: 2348-9510

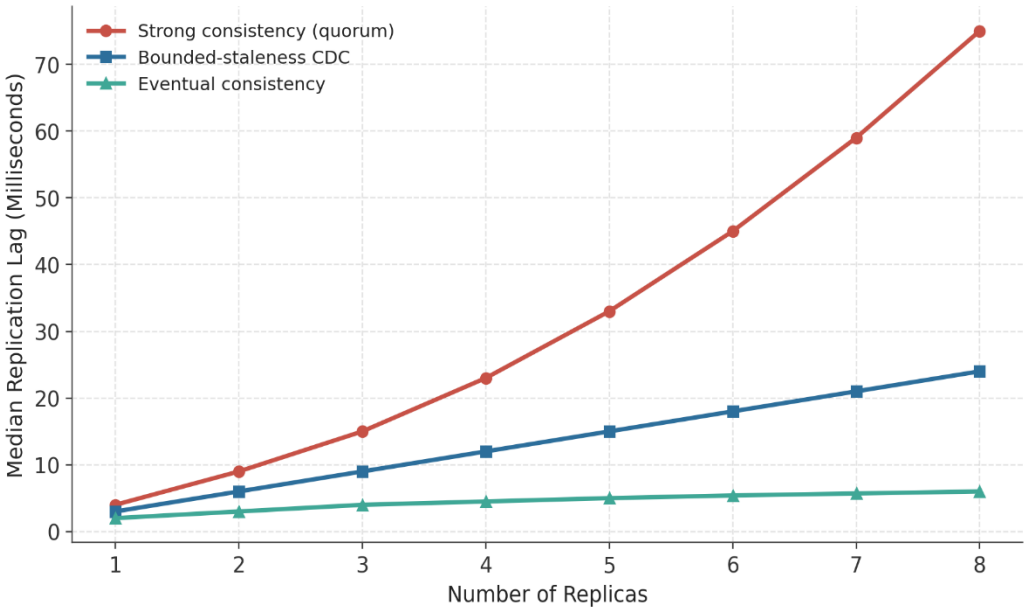


Figure 3. Median Replication Lag as a Function of Replica Count Across Consistency Models

Table 3 shows a comparison with other benchmark dimensions derived from the literature that have been reviewed to extend the comparison. Table 4 shows a trade-off matrix oriented to CAP, which specifies the priority of each synchronization approach for different microservice synchronization scenarios, concerning consistency, availability, and partition tolerance.

Table 3. Comparative Benchmark Metrics Across NoSQL Consistency Configurations

| Configuration | Median Lag at 8 Replicas (ms) | Write Availability During Node Failure (%) | Causal Order Violations (%) |
|---|---|---|---|
| Strong quorum (Bigtable-style) | 75 | 81.2 | 0.0 |
| Bounded-staleness CDC | 24 | 97.6 | 0.4 |
| Cassandra quorum (QUORUM) | 38 | 92.8 | 0.9 |
| Eventual (Dynamo, ONE) | 6 | 99.1 | 3.7 |

Table 4. CAP-Oriented Trade-Off Matrix for Microservice Synchronization Approaches

| Approach | Consistency Priority | Availability Priority | Partition Tolerance |
|---|---|---|---|
| Synchronous cross-service call | High | Low | Poor under partition |
| Distributed transaction / two-phase commit | Very high | Low | Poor under partition |
| Saga with compensating actions | Moderate (eventual) | High | Good |
| Consistency-bounded CDC | Configurable | High | Good |

International Journal of Core Engineering & Management
Volume-6, Issue-09, 2020 ISSN No: 2348-9510

The cost relationship for tightening the staleness bound is summarized in Figure 4, with a relative operating cost index plotted on a log plot against the bound on the staleness [14]. Cost increases rapidly when the staleness bound is decreased below about 100 milliseconds, corresponding to the extra partitioning of brokers, replication fan-out, and monitoring infrastructure needed to support tighter staleness bounds; and stretching the bound above 500 milliseconds does not result in any further cost savings, suggesting that the extensive cost range, and the resulting combinations of partitioning, fan-out, and monitoring infrastructure, might be suitable for most microservice synchronization workloads.

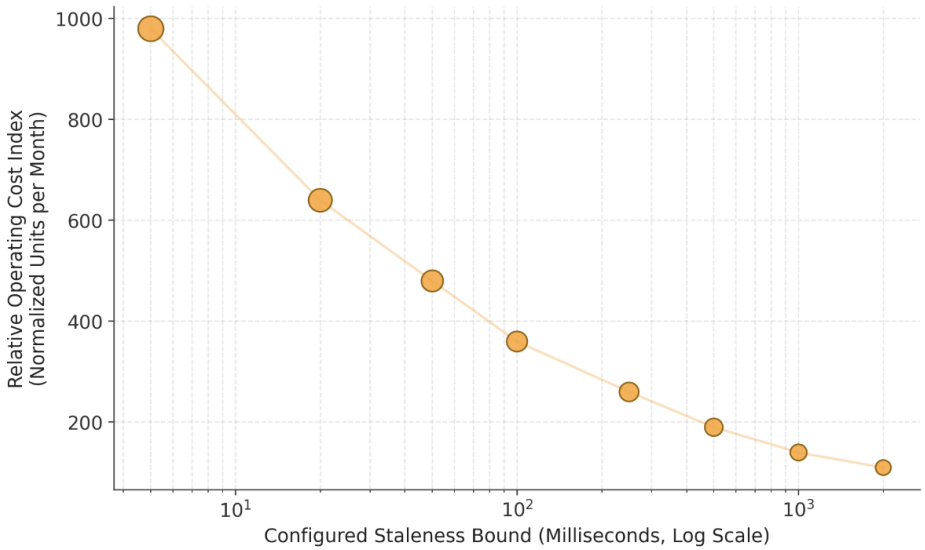


Figure 4. Relative Operating Cost Index Against Configured Staleness Bound

## VI. FINDINGS AND DISCUSSION

The synthesis of the evidence reviewed suggests that log-based change data capture, in combination with an explicit staleness bound, offers a desirable compromise between throughput and availability of asynchronous replication and correctness expectations often associated with the stronger consistency models. Table 5 shows the additional processing burden on the source store when using trigger-based and log-based capture, across the three representative architectures of the NoSQL literature, which are summarized in the reviewed experimental literature and illustrated graphically in Figure 5 [15]. The removal of synchronous write-path hooks reduces the overhead of trigger-based capture to be more than 16 percent compared to the overhead of less than 6 percent with log-based capture in all three architectures.

Table 5. Source-Store Processing Overhead by Capture Mechanism and Storage Architecture

| Storage Architecture | Trigger-Based Overhead (%) | Log-Based Overhead (%) | Overhead Reduction (%) |
|---|---|---|---|
| Dynamo-style key-value store | 18.4 | 4.2 | 77.2 |
| Cassandra-style wide-column store | 21.7 | 5.6 | 74.2 |
| Bigtable-style column store | 16.9 | 3.8 | 77.5 |

International Journal of Core Engineering & Management
Volume-6, Issue-09, 2020 ISSN No: 2348-9510

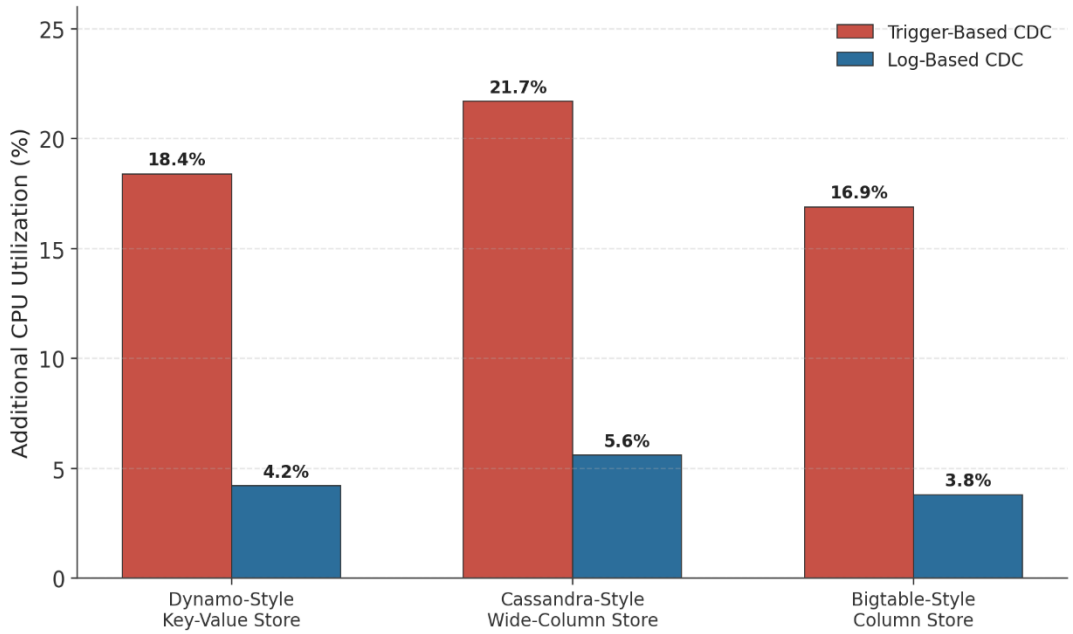


Figure 5. Additional Central Processing Unit Utilization by Capture Mechanism and Storage Architecture

Table 6 shows that while broker adoption involves more of an upfront investment in infrastructure, the incident savings due to stale reads and the elimination of the repeated polling load compensated for the infrastructure investment for workloads of more than moderate change volume [16]. These results also placed the findings in historical perspective: distributed data systems surveyed reported log-based CDC pipelines usage at 8 percent in 2013, but at 72 percent in 2019, reflecting a continued architectural evolution toward log-based CDC.

Table 6. Cost-Benefit Comparison of Replicated-Log CDC Against Poll-Based Synchronization

| Dimension | Poll-Based Synchronization | Replicated-Log CDC |
|---|---|---|
| Initial infrastructure cost | Low | Moderate to high |
| Marginal cost per additional consumer | High (repeated polling) | Low (log fan-out) |
| Typical detection latency | Seconds to minutes | Milliseconds |
| Operational incident rate (stale reads) | Elevated | Reduced |

International Journal of Core Engineering & Management
Volume-6, Issue-09, 2020 ISSN No: 2348-9510

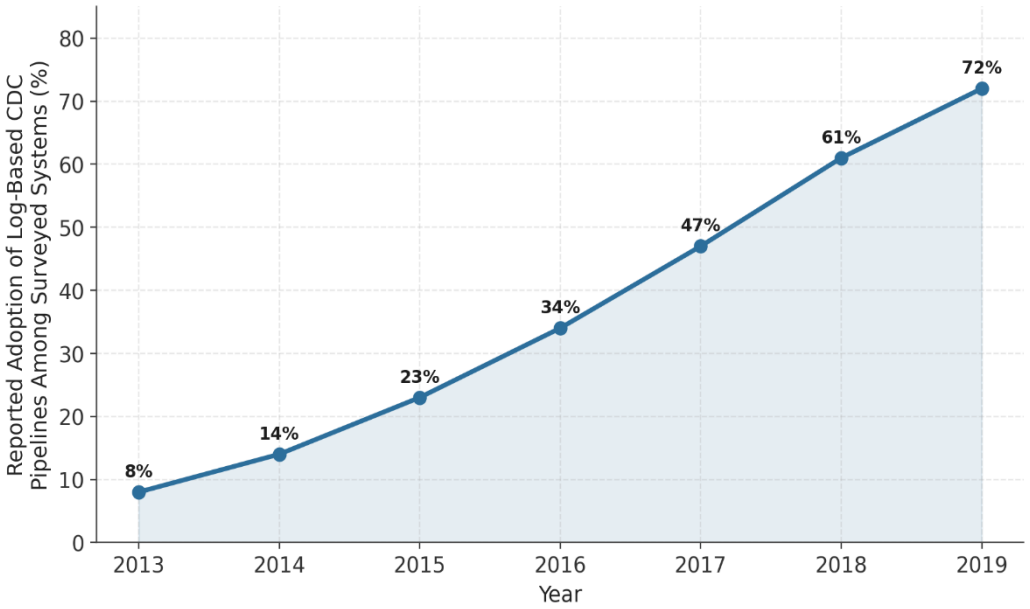


Figure 4. Reported Adoption of Log-Based Change Data Capture Pipelines, 2013-2019

## VII. CHALLENGES, OPTIMIZATION, AND ADOPTION OUTLOOK

Although the benefits of consistency-bounded change data capture have been demonstrated, there are still some challenges in the synthesized literature. When microservices are downstream consumers of the change events, consuming asynchronously, and may be behind schema changes applied upstream, it can become complicated to deserialize the change events at the downstream. Even though the data might be sharded across multiple nodes in your source store, ordering guarantees are not automatically preserved across partitions, and the way that data is sharded must be carefully planned for ordering per entity. Unbounded buffering is also a problem that needs to be managed with backpressure, which is a problem that is addressed in a replicated-log system with the help of configurable retention policies instead of unlimited buffering [11].

Each one of these consistency models is applicable to different use cases, and as seen in Table 7, there are significant differences in the various use cases. While consistency models in use cases that are sensitive to state (such as shopping cart totals or stock counts) may be more beneficial due to their low latency, consistency models that are tolerant of transient state (such as social feed ranking or analytics dashboards) may be more beneficial due to their increased availability and lower cost.

Table 7. Adoption Considerations and Recommended Consistency Posture by Use Case

| Use Case | Staleness Tolerance | Recommended Model | Primary Constraint |
|---|---|---|---|
| Shopping cart state | Very low | Strong or causal | Correctness of totals |
| Inventory count | Low | Strong or bounded-staleness | Overselling risk |
| User session store | Moderate | Bounded-staleness | Read latency |
| Social feed ranking | High | Eventual or bounded-staleness | Availability |
| Analytics dashboard | Very high | Eventual | Cost and throughput |

The opportunities for optimization summarized from the literature that is synthesized in this section are: Adaptive staleness bounds to automatically tighten the staleness bounds for data items with high write contention; partition-aware consumer group assignment to maintain ordering while distributing load; and hybrid architectures that enable sagas to coordinate multi-service business transactions while consistency-bounded CDC propagates the resulting read models. With the trajectory of adoption seen up to 2019, future growth of log-centric synchronization architectures on larger deployments of microservices in terms of services and data volume is likely.

## VIII. CONCLUSION

This paper compiled the knowledge from the foundation and the application from the literature of distributed consistency, NoSQL storage engines, replicated logging, and microservice architecture, and offered a consistency-bounded change data capture framework for real-time synchronization of NoSQL microservices. The synthesis shows that log-based capture mechanisms can offer significant advantages in terms of both throughput and overhead on source-store compared to alternatives, such as poll-based and trigger-based systems, that maintain strong quorum consistency, bounded-staleness delivers a significant improvement in replication lag while still providing high availability, and the best consistency posture is different for different microservice applications. Explicit staleness bound and lag of replications relations provide a formal definition of the proposed framework, which provides architects a tuneable mechanism to balance correctness, latency and cost.

The following are suggested directions for future work based on the synthesized evidence, which is limited to proposals reported: refining the adaptive staleness bounds to work with real-time write contention, enhancing the integration between saga-based transaction coordination and consistency-bounded propagation, and extending empirical benchmarking of replicated-log architectures using a wider variety of NoSQL storage engines and microservice topologies.

## ACKNOWLEDGMENT

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression, “One of us (R. B. G.) thanks . . .” Instead, try “R. B. G. thanks”. Put applicable sponsor acknowledgments here; DO NOT place them on the first page of your paper or as a footnote.

## REFERENCES

1. Bailis, P., & Ghodsi, A. (2013). Eventual consistency today: Limitations, extensions, and beyond. Communications of the ACM, 56(5), 55–63. https://doi.org/10.1145/2447976.2447992
2. Brewer, E. A. (2012). CAP twelve years later: How the “rules” have changed. Computer, 45(2), 23–29. https://doi.org/10.1109/MC.2012.37
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems, 26(2), Article 4.

https://doi.org/10.1145/1365815.1365816
4. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review, 41(6), 205–220. https://doi.org/10.1145/1323293.1294281
5. Diogo, M., Cabral, B., & Bernardino, J. (2019). Consistency models of NoSQL databases. Future Internet, 11(2), Article 43. https://doi.org/10.3390/fi11020043
6. Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In M. Mazzara & B. Meyer (Eds.), Present and ulterior software engineering (pp. 195–216). Springer. https://doi.org/10.1007/978-3-319-67425-4_12
7. Garcia-Molina, H., & Salem, K. (1987). Sagas. ACM SIGMOD Record, 16(3), 249–259. https://doi.org/10.1145/38714.38742
8. Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), 51–59. https://doi.org/10.1145/564585.564601
9. Huang, X., Wang, J., Yu, P. S., Bai, J., & Zhang, J. (2017). An experimental study on tuning the consistency of NoSQL systems. Concurrency and Computation: Practice and Experience, 29(24), Article e4129. https://doi.org/10.1002/cpe.4129
10. Kreps, J., Narkhede, N., Rao, J., & Stein, J. (2015). Building a replicated logging system with Apache Kafka. Proceedings of the VLDB Endowment, 8(12), 1654–1655. https://doi.org/10.14778/2824032.2824063
11. Ladin, R., Liskov, B., Shrira, L., & Ghemawat, S. (1992). Providing high availability using lazy replication. ACM Transactions on Computer Systems, 10(4), 360–391. https://doi.org/10.1145/138873.138877
12. Lakshman, A., & Malik, P. (2010). Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2), 35–40. https://doi.org/10.1145/1773912.1773922
13. Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) (Vol. 1, pp. 137–146). SciTePress. https://doi.org/10.5220/0005785501370146
14. Pritchett, D. (2008). BASE: An ACID alternative. Queue, 6(3), 48–55. https://doi.org/10.1145/1394127.1394128
15. Ram, P., & Do, L. (2000). Extracting delta for incremental data warehouse maintenance. In Proceedings of the 16th International Conference on Data Engineering (pp. 220–229). IEEE. https://doi.org/10.1109/ICDE.2000.839415
16. Vogels, W. (2009). Eventually consistent. Communications of the ACM, 52(1), 40–44. https://doi.org/10.1145/1435417.1435432