International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(18s) 2026 pp. 222-231
https://doi.org/10.70917/ijcisim-2026-4986
Article

# Decoupling Monolithic Databases: A Log-Based Change Data Capture Framework for Phased Microservices Evolution

**Girish Rameshbabu**

Customer Success Technical Architect, India, Email: girish.prasad.23@gmail.com

**Abstract:** The migration of established enterprise systems from monolithic architectures to microservices is frequently constrained not by application logic, which containerisation and orchestration render tractable, but by the data tier. Where multiple business domains operate against a shared relational schema, tight coupling at the data layer limits deployment independence and negates much of the agility that microservice decomposition is intended to deliver. Conventional remedies are unsatisfactory: batch Extract-Transform-Load introduces latency incompatible with operational coexistence, while application-level dual writing requires distributed transaction management and admits partial-failure states that are difficult to reconcile. This paper presents a framework for decomposing a monolithic data tier using log-based Change Data Capture (CDC) as the synchronisation mechanism, with the Confluent Platform and Kafka Connect serving as the reference implementation. The framework organises the migration into four sequential and individually reversible phases - passive shadowing, transformation and routing, dual-state reconciliation, and staged cutover - each aligned with the incremental substitution logic of the Strangler Fig pattern. We describe the implementation architecture spanning source connector configuration, schema governance via a registry-enforced data contract, and sink delivery to polyglot targets. We further analyse the principal trade-offs the approach entails, comparing log-based capture against the transactional outbox pattern, characterising the consistency window introduced by the shift from strong to eventual consistency, and addressing tombstone propagation, distributed tracing, and compensating transactions. Finally, we specify an evaluation protocol by which implementations of the framework may be empirically assessed. The framework is offered as a design reference for practitioners undertaking phased data-tier decomposition.

**Keywords:** change data capture; microservices migration; monolith decomposition; event streaming; Apache Kafka; schema evolution; eventual consistency; strangler fig pattern

## 1. INTRODUCTION

The transition from monolithic architectures to microservices is a significant undertaking for established enterprises. While application logic can be modularised through containerisation and orchestration, the underlying data layer often remains a singular, highly coupled entity. This centralised data structure poses substantial risks to modernisation efforts, as it limits the agility that microservices are intended to provide.

### *A. The Monolithic Bottleneck: Shared Database Constraints*

The primary challenge in architectural evolution is the shared database pattern. In a traditional monolith, multiple business domains - such as inventory management, financial processing, and logistics - operate against a common database schema. This configuration results in significant technical debt arising from tight coupling at the data tier. Any modification to a shared table may propagate unforeseen errors across unrelated modules, necessitating exhaustive regression testing and manual synchronisation of deployment cycles.

A shared database further imposes a vertical scaling limit. Independent services cannot scale their data resources according to their specific demand, producing inefficient resource utilisation and increased operational latency. The

database becomes the principal point of contention, constraining both deployment frequency and system resilience. Where multiple services access a single database instance, the risk of lock contention and resource exhaustion rises, degrading performance under peak load.

### *B. The Strangler Fig Pattern for Phased Decomposition*

To mitigate the risks associated with wholesale migration, organisations adopt the Strangler Fig pattern [5], [2]. This strategy facilitates the incremental replacement of legacy functionality by routing new features to microservices while retaining the existing system for legacy operations. Services are extracted one domain at a time, permitting a controlled transition that limits the blast radius of any individual failure.

The fundamental difficulty of this phased approach lies in state management. As functions migrate to new services, the corresponding data must be synchronised between the legacy environment and the new service-specific stores. Maintaining this dual state is essential if both the legacy monolith and the emerging microservices are to hold a consistent view of system data throughout what is typically a protracted transition. Absent a robust synchronisation mechanism, data silos emerge and produce inconsistencies that are difficult to reconcile retrospectively.

### *C. Problem Statement*

The central objective of this framework is to address the migration of data from a legacy Relational Database Management System (RDBMS) to modern distributed data stores without operational interruption. Conventional Extract, Transform, Load (ETL) methodologies are insufficient for this requirement, as they rely on periodic batch processing that introduces latency and windows of inconsistency incompatible with the coexistence of legacy and modern systems [23].

Application-level dual writing is likewise unsatisfactory. It requires complex distributed transaction management and admits data corruption during partial failure: where one write succeeds and its counterpart fails, the system enters an inconsistent state requiring manual intervention [14], [12]. There is consequently a need for a non-intrusive, log-based synchronisation mechanism capable of replicating state changes with minimal impact on source system performance. This paper examines log-based CDC connectors as the integration layer bridging these heterogeneous environments.

### *D. Contributions*

This paper makes four contributions:

1. A four-phase migration lifecycle for monolithic data-tier decomposition, in which each phase is individually reversible and verifiable before the next is entered (§IV).

2. An implementation architecture spanning source connector configuration, registry-enforced schema governance, and polyglot sink delivery (§V).

3. A trade-off analysis comparing log-based capture against the transactional outbox pattern, and a characterisation of the consistency window the approach introduces (§VII).

4. An evaluation protocol permitting comparable empirical assessment of framework implementations (§VIII).

## 2. RELATED WORK

### *A. Existing work relevant to this framework falls into three groups.*

Migration methodology. Chiu and Cheng propose a data migration framework for microservice modernisation, addressing the decomposition of legacy data stores as a staged process [1]. The Strangler Fig pattern, articulated by Fowler [5] and elaborated by Richardson [2], provides the incremental-substitution logic on which phased approaches rest. These treatments establish the sequencing of migration but do not specify the synchronisation mechanism by which dual state is maintained during the transition, which is the concern of the present work.

Change Data Capture mechanisms. The distinction between query-based polling and log-based capture is documented in both the practitioner and survey literature [23], [9]. Gupta et al. survey CDC in distributed systems, situating log-based capture within the broader integration landscape [9]. Debezium and its integration with Kafka Connect provide the engine on which most contemporary log-based connectors are built [8], [6]. This body of work characterises the capture mechanism itself; its application as the state-synchronisation layer of a phased architectural migration has received less systematic treatment.

Event streaming and data contracts. Kreps articulates the log as a unifying abstraction for real-time data integration [11], a framing developed at length by Kleppmann in the context of data-intensive system design [7] and by Narkhede et al. with respect to Kafka specifically [4]. Schema governance as a mechanism for enforcing compatibility across evolving consumers is addressed in [15] and [17]. The transactional outbox pattern, developed by Morling [22], represents the principal alternative to log-based capture and is compared against it in §VII.A.

The framework presented here draws on all three groups, and its contribution lies in their integration: it applies log-based capture as the synchronisation substrate for a Strangler Fig migration, governed by registry-enforced data contracts, and specifies the phase sequencing and reversibility conditions that such an integration requires.

## 3. METHOD

### *A. This paper develops a design framework rather than reporting controlled experiments. It was constructed in three stages.*

Stage 1 - Requirement derivation. The failure modes of existing approaches to data-tier migration - batch ETL, application-level dual writing, and query-based polling - were characterised from the literature reviewed in §II, and the requirements that a satisfactory mechanism must meet were derived from those failure modes. Three requirements emerged: non-intrusiveness with respect to the legacy source, full fidelity in capturing intermediate state, and reversibility at each stage of the transition.

Stage 2 - Phase construction. The migration lifecycle of §IV was constructed by decomposing the transition into the minimal sequence of states in which each successive state is independently verifiable and each transition is reversible without data loss. The four resulting phases are those presented in §IV.A through §IV.D.

Stage 3 - Trade-off analysis. Each design decision was examined for the constraints it imposes and the alternatives it forecloses, yielding the comparative analysis of §VII. Where a decision entails a consistency or operational cost, that cost is stated rather than minimised.

Scope of claims. The framework is analytical and is not validated by controlled measurement in this paper. Claims regarding mechanism behaviour are attributed to the cited sources, and the reader should note that a substantial proportion of the available literature in this area consists of vendor documentation and practitioner writing rather than peer-reviewed research. §VIII specifies the protocol by which the framework's claims may be tested empirically, and §IX states the limitations that follow from the absence of such testing here.

## 4. TECHNICAL FOUNDATION: LOG-BASED CHANGE DATA CAPTURE

The efficacy of the proposed framework rests on a transition from state-based data access to event-based data capture. Traditional integration methods treat the database as a static repository to be queried; log-based CDC treats it as a continuous source of discrete events, yielding a granular view of data evolution over time.

### *A. Log-Based versus Query-Based Capture*

Traditional data integration relies on query-based polling, in which a process periodically executes SELECT statements to identify new or updated rows by timestamp or incrementing identifier. This approach exhibits three deficiencies. It cannot readily identify deleted records, since a removed row simply ceases to appear in results. It imposes query load on the production database, competing with operational traffic. And it misses intermediate state

changes occurring between polling intervals, so that a row updated several times within one interval yields only its final value [23].

Log-based CDC operates instead at the storage engine layer. Most modern relational databases - including PostgreSQL, MySQL, and Oracle - maintain a Write-Ahead Log, Redo Log, or Binary Log: append-only files recording every committed transaction for recovery and replication purposes. Connectors intercept these logs, frequently by means of the Debezium engine [8]. In a MySQL environment, for instance, the connector registers as a replication client and receives the same binary log events a physical replica would.

This yields three technical advantages. The mechanism is non-intrusive, since reading the log avoids the CPU overhead and table contention associated with frequent polling. It provides full fidelity, capturing every intermediate state change rather than only the terminal value at each interval. And it handles deletions explicitly: the connector identifies the DELETE operation in the transaction log and propagates a tombstone event to downstream consumers [21].

### *B. Kafka as a Persistent Buffer*

Once captured, a change event is published to a topic within the event streaming platform. Kafka here serves not as a message queue but as a persistent, fault-tolerant transport layer that decouples the monolith from its consumers [4], [11].

Three properties are material to the migration use case. Backpressure management: during migration, the legacy database may produce events faster than a newly deployed microservice can process them; the log absorbs this differential by persisting events to disk and permitting the consumer to advance at its own rate without loss. Durability: should a microservice fail or its local store become corrupted, it may re-read the topic from an arbitrary historical offset to reconstruct its state without re-querying the legacy monolith. Exactly-once semantics: the platform's transactional guarantees prevent duplicate records in the target store during network partitions or connector restarts, a requirement that is acute in CDC scenarios where replay is routine.

Together these properties establish a durable source of truth in motion, decoupling the lifecycle of the monolithic data tier from that of the microservices consuming it.

## 5. Proposed Framework: The Evolution Lifecycle

The framework organises the transition into four phases. Each phase is designed to be individually verifiable and reversible, so that the organisation is at no point committed to an irrevocable cutover without validated synchronised data.

### *A. Phase 1: Shadowing*

The initial phase deploys a source CDC connector to shadow the legacy database. The connector observes transaction logs and streams every state change into topics. The process is entirely passive: the legacy application remains unaware of the extraction, and no modification to legacy schema or code is required.

Architecturally this constitutes a dark launch for data. It permits engineers to validate the event stream, monitor consumer lag, and confirm that the event envelope - including timestamps and transaction identifiers - is captured correctly. Establishing a throughput baseline without affecting the production workload allows the streaming infrastructure to be sized appropriately before any dependency is created upon it.

The reversibility condition for this phase is trivially satisfied: the connector may be removed with no effect on the source.

### *B. Phase 2: Transformation and Routing*

Monolithic schemas are typically normalised for storage efficiency or shaped by legacy constraints rather than by business domain boundaries. This phase reshapes raw database events into domain-aligned messages suitable for consumption by specialised services, by two mechanisms.

Single Message Transformations are applied within the Connect runtime before data is written to the topic. They accommodate lightweight, stateless logic: masking sensitive fields to satisfy compliance requirements, filtering columns irrelevant to downstream consumers, or routing events to specific topics on the basis of field values [13].

Stream processing addresses transformations requiring state or joins. Monolithic data is frequently distributed across multiple tables - order headers and order line items, for example - whereas the consuming service may require a single flattened domain event. Stream processing performs the necessary joins and aggregations on the incoming streams so that the service receives an event matching its own domain model rather than the source schema.

Reversibility here derives from the immutability of the source topic: transformation logic may be revised and the stream reprocessed from the original events.

### *C. Phase 3: Dual State and Reconciliation*

During the interim state, monolith and microservice coexist. This phase addresses the dual-write problem, in which an application writing to two systems simultaneously produces inconsistency when one write succeeds and the other fails [14].

The framework eliminates application-level dual writing entirely. The legacy monolith remains the sole writer, and the microservice store is updated asynchronously by a sink connector consuming from the topic. Because the write path is singular, the partial-failure mode that characterises dual writing cannot arise; failures instead manifest as consumer lag, which is observable and recoverable.

To ensure integrity, a reconciliation loop is established in which the microservice periodically compares a checksum of its local state against the streamed source of truth. This permits detection of drift and confirms that processing delays in the sink do not become permanent divergence [12].

### *D. Phase 4: Cutover*

The final phase shifts the source of truth. Once the microservice demonstrates stability and minimal synchronisation lag, application traffic is redirected. Two strategies are recommended.

Under canary cutover, a bounded proportion of write traffic - segmented by user identifier or region - is directed to the new service. During this period the capture direction is reversed, streaming changes from the microservice store back to the legacy monolith. This preserves the legacy system as a viable fallback should the new service fail under load, and it is the reversibility condition for this phase.

Under final decommissioning, once the legacy system no longer receives write traffic, its tables are placed in read-only mode for audit purposes, the source connector is disabled, the reverse stream is stopped, and the microservice becomes the sole authority for its domain.

## 6. IMPLEMENTATION ARCHITECTURE

*Execution of the framework rests on three components: source connectors, schema governance, and sink connectors.*

### *A. Source Configuration and RDBMS Interfacing*

Implementation begins with source connectors specific to the legacy RDBMS. Unlike generic JDBC connectors, which poll, these interface directly with the database’s native replication protocols. An Oracle implementation may employ LogMiner or XStream to extract changes from the Redo Logs; a PostgreSQL implementation leverages logical replication slots and the pgoutput plugin to stream from the Write-Ahead Log.

Configuration of snapshot mode is critical. An initial consistent snapshot of existing data must be taken before the transition to incremental log-based capture, so that the downstream service holds a complete record of domain state rather than only changes occurring after connector deployment. The boundary between snapshot and incremental capture is the point at which correctness is most readily compromised, and it warrants explicit verification.

### *B. Schema Governance and Data Contracts*

In a monolithic environment schema changes are frequently applied manually, which can produce failures in downstream consumers when the data format shifts unexpectedly. The architecture therefore integrates a schema registry.

When a connector captures a change, the schema of the source table is registered, and data is serialised using Avro or Protobuf. These formats are preferred over plain JSON because they are compact, support rich type systems, and enforce compatibility rules - backward, forward, or full - across schema versions [15]. This constitutes a formal data contract between monolith and microservices: an attempt to drop a column on which a consumer depends can be blocked at the producer or surfaced to the consumer before it manifests as a runtime failure [17].

### *C. Sink Connectors and Polyglot Persistence*

The final stage delivers transformed events to service-specific stores, permitting a shift from a uniform relational monolith to polyglot persistence in which each service adopts a store suited to its query patterns.

A service managing product catalogues may employ a document store, with flat relational records transformed into nested documents supporting schema flexibility the source RDBMS could not provide. A service concerned with relationship traversal - recommendation or fraud detection - may employ a graph store, with nodes and edges constructed as source transactions occur. A service requiring full-text retrieval may employ a search index maintained continuously from the same stream [16].

Because ingestion is decoupled from egress by the log, multiple sinks may consume the same source topic, permitting simultaneous population of search indexes, analytical stores, and operational service databases without additional load on the source.

## 7. TRADE-OFF ANALYSIS AND RISK MITIGATION

### *A. Log-Based Capture versus the Transactional Outbox*

The principal design alternative is the transactional outbox pattern, in which the legacy application is modified to write events into a dedicated table within the same local transaction as the business data, thereby guaranteeing atomicity between state and event [22].

Where the legacy monolith can be modified safely, the outbox pattern offers stronger guarantees and independence from the database's log structure and retention policy. Where code changes are high-risk or the source is effectively opaque, log-based capture is preferable: it requires no application modification and avoids the operational burden of an outbox table. The trade-off is a dependency on log retention configuration and a requirement for administrative access to the database engine [23]. The choice is therefore governed by the modifiability of the legacy source rather than by the intrinsic superiority of either mechanism.

### *B. Consistency Model and the Inconsistency Window*

Adopting CDC entails moving from strong consistency within the monolith to eventual consistency across the resulting distributed system. The inconsistency window - the interval between commit in the legacy RDBMS and visibility in the consuming service - is determined by network latency, ingestion lag, and consumer processing time [24]. Its magnitude is a property of a particular deployment and must be measured rather than assumed; §VIII specifies its measurement.

The architectural consequence is that consuming services must be designed to tolerate this window. This requires idempotent consumers, so that duplicate delivery does not corrupt local state, and version or timestamp ordering, so that out-of-order arrival is detected rather than silently applied. Services whose correctness requires read-your-writes semantics against the migrated domain are poor candidates for this approach and should be identified before migration rather than during it.

### *C. Compensating Transactions*

Where a consuming service fails to process an event already committed in the monolith, the systems diverge. Because the source transaction is final and cannot be rolled back, the framework employs compensating transactions within a Saga structure [25].

On sink failure, an orchestrator or stream processor detects the condition and emits a compensating event. Where an order-created event fails to reach a billing service, for example, a compensating event is directed to a reconciliation topic, notifying the legacy system or recording the discrepancy for manual resolution, so that the two systems converge.

### *D. Handling Hard and Soft Deletes*

*Deletion is handled differently according to its implementation in the source.*

Soft deletes, in which the application sets a flag column, appear to CDC as ordinary update events. They are handled by filtering in the stream processor or by propagation as explicit deactivation events.

Hard deletes, in which the row is physically removed, are detected by the connector in the transaction log, which emits a tombstone: a message bearing a valid key and a null value. The tombstone signals log compaction to remove the key and instructs the sink to delete the corresponding entry downstream [19], [21].

Failure to handle hard deletes correctly produces residual records that persist in the consuming service after purge from the monolith - a data integrity defect with compliance implications where the deletion was performed to satisfy a retention or erasure obligation.

### *E. Distributed Tracing*

In a monolith, tracing a transaction is straightforward, as it occurs within a single process boundary. Under CDC a transaction spans the source database, the connector, the log, and the sink. End-to-end visibility therefore requires distributed tracing.

Instrumenting the Connect runtime with OpenTelemetry injects trace context headers into each change event; trace and span identifiers are preserved in message headers as the event traverses the log. This permits reconstruction of a record’s full lifecycle, from commit in the source to materialisation in the consuming store [18], and is the practical prerequisite for diagnosing both latency and apparent event loss in high-throughput deployments.

### *F. Schema Drift*

Schema drift - a change in the legacy database that breaks downstream consumers - is a persistent risk. The framework mitigates it through registry-enforced compatibility modes. Transitive compatibility settings ensure that a change in the monolith remains compatible with all prior schema versions, so that evolution does not require simultaneous redeployment across all consuming services.

Emerging work explores machine-learning approaches to predicting the impact of schema changes and proposing mappings between heterogeneous systems [20]. Such approaches are not yet established practice, and the framework as specified relies on registry-enforced compatibility rather than on predictive adaptation.

### *G. Operational Model: Managed versus Self-Hosted*

The choice between managed and self-hosted deployment affects the operational burden borne by the engineering organisation.

A managed service abstracts the infrastructure layer - broker management, consensus configuration, and connector scaling - allowing teams to concentrate on transformation logic and domain modelling [26]. A self-hosted platform offers finer control and permits data residency within private infrastructure, at the cost of dedicated capacity for patching, tuning, and capacity planning [27]. The determining factors are regulatory data-residency constraints and the availability of platform engineering capacity, rather than cost alone.

## 8. EVALUATION PROTOCOL

The framework is presented analytically and is not empirically validated here. This section specifies a protocol by which implementations may be assessed, so that results from independent deployments are comparable.

Source workloads. Assessment should span at least three write profiles: a steady low-rate transactional profile; a bursty profile exhibiting order-of-magnitude rate variation; and a bulk profile representing batch operations against the legacy source. Each should be characterised by transaction rate, average transaction size, and the proportion of updates and deletes.

Baselines. The framework should be compared against scheduled batch ETL and against query-based polling, on identical workloads, so that the contribution of log-based capture is distinguishable from that of the streaming transport.

Metrics. End-to-end propagation latency at the 50th, 95th, and 99th percentiles, measured from source commit to sink visibility, which operationalises the inconsistency window of §VII.B; source system overhead, measured as CPU and I/O attributable to capture; consumer lag under each workload profile; delete propagation completeness, measured as the proportion of source deletions correctly reflected downstream; and reconciliation drift, measured as the divergence detected by the checksum loop of §V.C over a sustained run.

Failure injection. Because the framework’s claims concern resilience, assessment should include connector restart during active capture, network partition between connector and broker, and sink unavailability exceeding the consumer timeout, verifying in each case that no committed source change is lost and that no duplicate is durably applied.

Reporting. Results should state source database engine and version, connector and platform versions, hardware configuration, retention and compatibility settings, and whether measurements derive from a production or synthetic workload.

## 9. LIMITATIONS

*Four limitations bound the claims of this paper.*

First, the framework is not empirically validated. The phase sequencing of §V and the reversibility conditions asserted for each phase are argued from the structure of the transition rather than demonstrated by measurement. §VIII exists because that validation remains outstanding.

Second, the available literature in this area is weighted toward vendor documentation and practitioner writing rather than peer-reviewed research. Where claims rest on such sources, they should be read as reports of engineering practice rather than as independently replicated findings.

Third, the framework assumes that the legacy source exposes a transaction log and that administrative access to it is obtainable. Legacy systems that do not satisfy these conditions - including certain managed database services and older engines without logical replication support - fall outside its scope, and for these the outbox pattern of §VII.A remains the applicable alternative.

Fourth, the treatment addresses the mechanics of data synchronisation and not the identification of service boundaries. Determining which domains should be extracted, and in what order, is a prerequisite to applying this framework and is not addressed by it.

## 10. CONCLUSION AND FUTURE WORK

### *A. Summary*

The transition from monolithic architecture to microservices carries risks of data loss, service interruption, and state inconsistency that are concentrated in the data tier rather than the application layer. This paper has argued that log-based Change Data Capture provides a non-intrusive mechanism for managing that transition, and has presented a four-phase framework applying it.

Three conclusions follow. On operational continuity, log-based capture enables continuous synchronisation without the latency of batch ETL or the source-system overhead of query-based polling, because it reads a log the database already maintains. On data integrity, registry-enforced schema governance converts the implicit coupling of a shared schema into an explicit data contract, so that incompatible changes are surfaced at the point of change rather than as downstream runtime failures. On architectural flexibility, the log as an intermediate buffer permits multiple heterogeneous sinks to consume a single source stream, enabling polyglot persistence without proportional load on the legacy system.

The framework's central design property is that each phase is reversible: at no point before final decommissioning is the organisation committed to a state from which it cannot retreat.

### *B. Future Directions*

Three directions follow. The most immediate is empirical validation under the protocol of §VIII, and in particular measurement of the inconsistency window of §VII.B across representative workloads, since that window is the principal cost the approach imposes and is currently characterised only qualitatively.

The second concerns automation of the migration itself. Agents that monitor traffic patterns and data dependencies within a monolith might identify suitable extraction boundaries and cutover points, and scale capture capacity in response to observed ingestion load, reducing the manual effort the framework presently requires.

The third concerns schema mediation. Machine learning approaches to generating transformation logic by analysing semantic relationships between legacy relational schemas and modern document models have been proposed [20], and their extension toward pipelines capable of applying versioned transformations automatically when incompatibility is detected represents a natural continuation. Such capability would reduce, though not eliminate, the manual intervention that schema evolution currently demands.

**REFERENCES**


1. H. J. Chiu and Y. C. Cheng, "A data migration framework for microservice modernization," Journal of Information Science and Engineering, vol. 38, no. 5, pp. 915–932, 2022.
2. C. Richardson, "Strangler fig application pattern: Incremental modernization to services," Microservices.io, June 2023.
3. S. Kumar and R. Singh, "Real-time data streaming and integration using Confluent Kafka," International Journal of Research Publication and Reviews, vol. 6, no. 5, 2025.
4. N. Narkhede, G. Shapira, and T. Palino, Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale, 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
5. M. Fowler, "StranglerFigApplication," martinfowler.com, 2004.
6. Confluent, "Change data capture (CDC) with Kafka Connect," Confluent Documentation, 2024.
7. M. Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol, CA: O'Reilly Media, 2017.
8. G. De Maio and J. Weichel, "CDC and data streaming: Capture database changes in real-time with Debezium," Confluent Blog, 2023.
9. A. Gupta et al., "Scalable real-time data integration: A survey of change data capture in distributed systems," arXiv preprint arXiv:2512.16146, 2025.

10. Confluent, “Why CDC with Kafka is the foundation for scalable microservices,” Confluent Blog, 2024.
11. J. Kreps, “The log: What every software engineer should know about real-time data’s unifying abstraction,” LinkedIn Engineering, 2013.
12. Solace, “Solving the dual-write problem with CDC and micro-integration,” Solace Blog, 2023.
13. G. Morling, “Single message transforms: The Swiss Army knife of Kafka Connect,” Gunnar Morling’s Blog, 2022.
14. Confluent, “The dual-write problem: Why CDC is the better way,” Confluent Blog, 2024.
15. J. Kim, “Schema registry part 1: Introduction to schema management,” Jaehyeon’s Tech Blog, 2022.
16. Confluent, “CDC and streaming analytics using Debezium and Kafka,” Confluent Blog, 2023.
17. Monday Systems, “Kafka schema registry in microservices: Ensuring data consistency,” MondaySys Blog, 2023.
18. Last9, “Distributed tracing for Kafka with OpenTelemetry,” Last9 Blog, 2024.
19. BinaryScripts, “Data integrity and risk mitigation in legacy systems,” SearchHounds Technology Series, 2024.
20. P. Sharma, “AI-driven change data capture (CDC) in BigQuery vs. traditional databases: A comparative analysis,” International Journal of Intelligent Systems and Applications in Engineering, vol. 11, no. 3, 2023.
21. Estuary, “Handling hard deletes in log-based CDC pipelines,” Estuary Engineering, 2024.
22. G. Morling, “Reliable microservices data exchange with the outbox pattern,” Debezium Blog, 2019.
23. Estuary, “Log-based CDC vs. traditional ETL: Key differences,” Estuary Blog, 2023.
24. Microsoft, “Saga design pattern for distributed transactions,” Azure Architecture Center, 2024.
25. C. Richardson, “Pattern: Saga,” Microservices.io, 2024.
26. Seal.io, “Reduce cognitive load in software engineering through platform engineering,” Dev.to, 2024.
27. AutoMQ, “Confluent Cloud vs. Confluent Platform: Architectural comparison,” AutoMQ Blog, 2024.