The Eastasouth Journal of Information System and Computer Science
Vol. 1, No. 03, April 2024, pp. 257-266
ISSN: 3025-566X, DOI: 10.58812/esiscs.v1i03

Schema-Evolution-Safe Polyglot Persistence for Containerized Microservices Using Event-Carried State Transfer: A Literature Synthesis

Srinivas Nune
Independent Researcher, Franklin, USA

Article Info

Article history:

Received, April, 2024
Revised, April, 2024
Accepted, April, 2024

Keywords:

CAP Theorem;
Container Orchestration;
Distributed Transactions;
Event-Carried State Transfer;
Event-Driven Architecture;
Eventual Consistency;
Microservices;
NoSQL;
Polyglot Persistence;
Saga Pattern;
Schema Evolution;
Schema Registry

ABSTRACT

Polyglot persistence is the new approach for container-orchestrated microservices, where each microservice chooses a data-store technology that best fits its workload; and event-carried state transfer, where state changes are carried as immutable events through a shared log, instead of directly through synchronous calls between services. This literature survey investigates the interplay of three research lines: schema evolution in heterogeneous NoSQL and polyglot data stores, event-driven integration patterns to minimize synchronous coupling and distributed transaction management by the saga pattern. The review is based on the distributed-systems theory, empirical microservice-migration studies, and schema-evolution measurement studies, to synthesize a conceptual framework that enables safe, independently evolvable microservice data layers through schema-registry-governed compatibility checking, event-carried state transfer, and compensation-based transaction management. The synthesis emphasizes the classic dilemma of consistency vs availability, as summarized by the CAP theorem, as well as the operational rigidity of production database schemas, and the flexibility that NoSQL stores with schema-on-read are offered but not guaranteed. Four conceptual diagrams and three comparative tables are used to summarize the reviewed material as an integrated architectural viewpoint, whereas two formal notations are used to summarize the consistency and transaction-compensation reasons that underlie the framework. The review finds that schema governance, compatibility contracts and compensating transactions are complementary mechanisms that do not compete and finds open challenges around tooling maturity, consistency on cross-store data and schema-level lifecycle management at the container-orchestration level.

Corresponding Author:

Name: Srinivas Nune
Institution: Independent Researcher, Franklin, USA
Email: Srinivas.nune9792@gmail.com

1. INTRODUCTION

The trend of breaking monolithic applications into self-contained microservices has been reported as a major architectural reaction to the scalability and maintainability issues of large, tightly coupled codebases [1], [2]. The trait that, as described by systematic mapping of the architecting literature, distinguishes the style from others is that each

service is expected to have its own data and expose that data through well-defined interfaces; historically this has led to a style of developing services, where different data-storage technologies are used to access different services, depending on the patterns they follow, and not necessarily the same relational schema is used across the whole system, which is known as polyglot persistence [3], [4].

This pattern is widely reported as being made feasible at scale by the use of a container orchestration platform, which enables each service, including its own private data store, to be packaged, scheduled, and scaled as an independent unit [5]. But it does not mean services are independent of each other and can't share state. Decomposition has been the cause of the large shift towards event-driven integration where services publish events describing state changes and other services consume them asynchronously [6], [7]. The pattern is usually called event-carried state transfer because with the full or partial state that comes with the event, consuming services can have local, denormalized versions of the data they need, without direct queries to the producing service.

This design provides a solution for one coupling issue and raises another. After a service consumes a locally generated schema from a stream of events produced by other services, all consumers of the service are now dependent on the structure of the events, which, by construction, changes when the event schema changes. Empirical research on the evolution of schemas in NoSQL and polyglot systems shows that after the schema has been adopted, it tends to be rigid and evolves at irregular intervals, implying that having no enforced relational schema is not enough to resolve the coordination problem schema evolution brings to a distributed system [8], [9]. Meanwhile, distributed-systems theory dictates that there are no data-store designs which can simultaneously offer strong consistency, availability and network partition tolerance, so any polyglot design that crosses stores with different consistency guarantees must make explicit decisions about where staleness is acceptable and where it is not [10], [11].

This review aims at providing a synthesis of these three, mostly disjointed, literatures: polyglot persistence and microservice data ownership, schema evolution in NoSQL / event-carried data and distributed transaction management by means of the saga pattern, and to provide a conceptual model of how to organize a schema-evolution-safe polyglot persistence using event-carried state transfer. The rest of the paper is structured as follows. In Section 2, material from the literature on each of the three threads is reviewed. The synthesis method employed to integrate the sources are described in Section 3 and the formal notations that support the discussion in Section 4 are presented. The integrated architectural framework is introduced in Section 4 along with an analysis of the tensions found in the sources analysed. The synthesis is summarised in section 5 and the limitations from the literature are summarised.

2. LITERATURE REVIEW

2.1 Microservice Decomposition and Polyglot Persistence

The microservices concept gained popularity as a description of an architecture that consists of a collection of small, independently deployable services, that run their own processes and communicate with each other via lightweight communication mechanisms [1]. Newman focuses on the style by pointing out that independent deployability demands independent data ownership, a shared data base for services being coupled, and thus lost [2]. This focus is confirmed by a systematic mapping study of the architecting literature that was conducted, which reveals that decentralized data management is one of the most common architectural issues discussed in the empirical microservices literature, along with granularity of services and inter-service communication style [12]. Early service-based practices have been cited as

the source of the style's evolution and the use of containerization and orchestration as enabling technologies for fine-grained per-service infrastructure to be economically viable has been recorded [4].

The concept of Polyglot persistence, that is using a different kind of storage technology for each of the various parts of an application rather than the default of a single relational engine, was defined as a named strategy in the NoSQL literature, where non-relational stores are organized into different families including key-value, document, column-family, and graph stores, each optimized for a different type of access patterns [3], [13]. Empirical case studies suggest that microservices migrations of monolithic single-schema databases are motivated mainly by the requirement to match the storage model to the query patterns of the microservices, as described in the multi-model polyglot layout within service boundaries [14].

In a complementary review of polyglot persistence patterns in microservice-based apps, the authors summarize the pros and cons of per-service, shared, and hybrid designs, and suggest that per-service ownership is preferable when independent scalability and technology fit outweigh the operational costs of dealing with heterogeneous stores [15]. Container orchestration surveys place this operational cost into the context of Resource scheduling, Service discovery and Lifecycle management, and how orchestration platforms hide a lot of the heterogeneity of underlying compute and storage resources, but are not a solution to data consistency issues between different services [5].

2.2 Schema Evolution in NoSQL and Polyglot Data Stores

As noted in empirical studies of schema-evolution, deployed database schemas evolve significantly less often and much less gracefully than is often suggested by flexible data models. A large scale empirical analysis of relational schema histories revealed that schema change is mostly episodic, consisting of long periods of stability that are followed by occasional change, often in disruptive ways, which the authors term as the gravitation toward rigidity [8]. An empirical investigation of the practices used for designing and evolving schemas for NoSQL document stores reveals similar trends in a schema-flexible system: While the underlying engines are not schema enforcing, schemas do emerge and evolve at the application level, and this hidden schema evolution is less easy to see and more difficult to control, given that the schemas are not enforced by the system.An empirical study of the practices adopted for designing and evolving schemas in NoSQL document stores confirms the same patterns: schemas evolve and appear at the application level without being enforced by the system, and this “hidden” schema evolution is not as easy to observe and control, as it is not enforced by the system itself [9].

To build on this observation, researchers have investigated explicit schema versioning and migration strategies, such as eager migration (rewriting existing records to support a new version of the schema) and lazy migration (iteratively migrating records as they are accessed), along with hybrid strategies of a combination of both to balance migration costs with the staleness of unmigrated records [16]. Building on this, self-adapting migration strategies suggest that migration strategy should be dynamically selected based on observed access patterns and workload properties, instead of being pre-determined [17]. These mechanisms were built mainly for a single NoSQL store, and the event-driven microservice architecture makes this situation more complex, since if a schema change is made in one microservice's local store, it propagates, via published events, to all consuming microservices' local stores, resulting in a

stream of more to more schema changes that have to be coordinated across all microservices involved in a logical schema change.

2.3 *Event-Driven Architecture, Consistency, and Event-Carried State Transfer*

As described in the microservices literature, event-driven architecture is an integration pattern that describes a scenario in which services communicate with each other by generating and consuming events through a persistent, ordered event log, instead of making direct calls on each other [6]. The style is believed to have the advantage of reducing temporal coupling (a consumer doesn't have to be ready to receive the event at the time it is produced), and structural coupling (a producer doesn't have to know about the identities of the consumers). This systematic treatment of data-intensive system design places event logs as a generic way to propagate state changes and defines a pattern for how downstream services can create their own local, denormalized representations continuously while feeding on an upstream event stream—event-carried state transfer [7].

As event-carried state transfer must be asynchronous, the consistency guarantees that consumers can count on are constrained by the event propagation delay, which can be captured by the impossibility result showing that a distributed system cannot simultaneously guarantee strong consistency, availability, and tolerance of network partitions [11]. Further research on eventual consistency explores the practical compromise between strict consistency and unconstrained availability, offering a probabilistic characterization of how fast the system can converge to consistency following a write and cataloguing methods, such as session guarantees and read-your-writes semantics, that restrict the practical impact of staleness without compromising on availability [10]. Comparative analyses targeting the practitioner position the eventual-consistency and strong consistency trade-offs as a per data type/per use case architectural decision rather than an architectural commitment that applies to the entire microservices system, as it is common for services to have different staleness tolerances within a microservices system [18].

2.4 *Distributed Transaction Management: The Saga Pattern*

Long-lived transactions that involve multiple independently managed data stores are not able to use a single atomic commit protocol without bringing in unnecessary service coupling. The saga concept, designed for "long-lived transactions" in centralized database systems, overcomes this by breaking the logical transaction into a series of local transactions that are each "compensated" by a compensating transaction that can undo the effects of the previous transaction in the sequence if the following transaction fails [19]. In the context of microservices, this pattern is described as the typical way to ensure data consistency between services without relying on distributed locks (orchestration and choreography are the two main styles of coordination) [20].

Later studies have tried to overcome some of the flaws of the basic saga formulation. One study suggests improvements that would minimize the operational complexity of matching compensating transactions between services, including cases where compensations can fail, or multiple sagas use shared resources [21]. Another line of work continues the saga and the wider microservice patterns and poses that compensation logic should be a first-class citizen rather than an additional afterthought on top of the main transaction logic [22]. Distributed transaction management is a challenge that is identified in its own right from empirical studies of the phenomenon of migrating to microservices, and the absence of the familiar relational transactional guarantees is especially

highlighted in the many challenges that organizations report as they migrate to microservices [23].

Table 1. Comparative Overview of Polyglot Data-Store Categories Discussed in the Reviewed Literature

Store category	Representative access pattern	Typical consistency posture	Source(s)
Relational	Structured, multi-entity queries with enforced integrity constraints	Strong, transactional	[3]
Key-value	High-throughput lookup by a single key	Tunable; often eventual	[3], [13]
Document	Nested, semi-structured records read as a unit	Tunable; schema emerges at application level	[9], [13]
Column-family / columnar	Wide, sparse rows; analytical or time-series access	Tunable; write-optimized	[13]
Graph	Traversal of densely interconnected entities	Typically strong within a partition	[13]

Source: Categories and characterizations synthesized from the NoSQL and polyglot-persistence literature reviewed in Section 2.1.

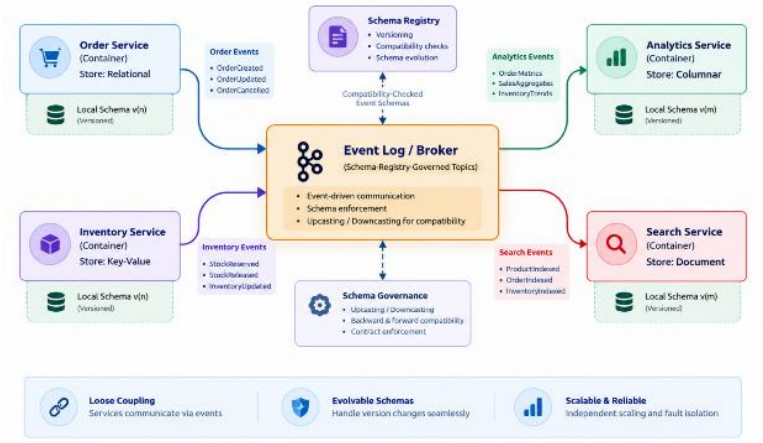

Figure 1. Event-Carried State Transfer Across Containerized, Polyglot-Persistent Microservices
Source: Each service retains a private, independently versioned local schema; the broker enforces compatibility on published event schemas.

3. METHODS

The synthesis which is reported in this paper is a narrative literature-review approach that considers the conceptual and cross-disciplinary nature of the question being addressed. These sources were selected from four related literatures: basic distributed-systems theory, distributed transaction/ Saga pattern theory, empirical and prescriptive microservices-architecture literature, and NoSQL and polyglot-persistence schema-evolution studies. The key architectural claims, empirical results, and mechanisms proposed in each source were identified, and then compared across the other three literatures to find similarities, conflicts, and missing connections. A summary of the sources that have been reviewed and how they correspond with the four literatures synthesized is provided in Table 2.

Table 2. Mapping of Reviewed Sources to the Four Synthesized Literatures

Literature thread	Reviewed sources
Microservices architecture and decomposition	[1], [2], [4], [5], [12], [23]

Literature thread	Reviewed sources
Polyglot persistence and NoSQL data models	[3], [13], [14], [15]
Schema evolution in NoSQL and polyglot stores	[8], [9], [16], [17]
Consistency theory and event-driven integration	[6], [7], [10], [11], [18]
Distributed transactions and the saga pattern	[19], [20], [21], [22]

The two formal notations that are repeated throughout the synthesized literature and restated here to aid the discussion in Section 4 are the following. The two formal notations, which are repeated throughout the synthesized literature and restated here to aid the discussion in Section 4 are the following: The impossibility result for distributed data stores proves that it is impossible to maintain strong consistency and full availability in a system when it experiences a network partition. This trade-off is expressed in Equation (1) where C represents strong consistency, A represents availability, and P represents partition tolerance, and P stands for the fact that the conjunction of all three cannot be satisfied [11].

$$C \wedge A \wedge P \Rightarrow false \quad (1)$$

Second, the probabilistic treatment of eventual consistency expresses the probability that a read observes a stale value as a decreasing function of the elapsed time since the corresponding write, parameterized by the replication-delay distribution of the underlying store. A simplified form of this relationship, following the probabilistic bounded-staleness formulation, is given in Equation (2), where P denotes the probability that a read occurring Δ time units after a write returns a value older than that write, and F is the cumulative distribution function of the store's inter-replica propagation delay [10].

$$P_{stale}(\Delta) = 1 - F(\Delta) \quad (2)$$

Equation (2) formalizes the intuitive property, used throughout the eventual-consistency literature, that as the elapsed interval Δ grows, the probability of staleness falls toward zero once the replication delay distribution F has fully propagated the write, and that the shape of F, rather than a single fixed number, determines how quickly a given polyglot store converges [10]. Third, the saga pattern is formalized as a sequence of local transactions with a matching sequence of compensations. Given a saga consisting of local transactions T_1 through T_n, if transaction T_k fails, the compensations C_{k-1} through C_1 are executed in reverse order to semantically undo the effects of the transactions that already committed, as expressed in Equation (3) [19], [20].

$$Saga = (T_1, \ldots, T_{k-1}, T_{k\,(fails)}) \Rightarrow (C_{k-1}, \ldots, C_1) \quad (3)$$

4. RESULTS AND DISCUSSION

The combination of the four literatures reviewed in Section 2 results in a layered framework: per-service data-model fit (provided by the polyglot perspective); event-carried state transfer (as the integration mechanism); schema registry (as the governance layer that prevents incompatible changes in the event-carried data across independently evolving services); and the saga pattern (as the mechanism for maintaining cross-service transactional integrity when a single logical operation is carried out across more than one service). Figure 1 shows the resulting architecture, Figure 2 shows the CAP trade-off determining the consistency posture each service/each store must assume, Figure 3 shows the schema-compatibility workflow that determines whether or not the event schema may change, and Figure 4 shows the saga compensation flow that determines how a partially completed cross-service operation is unwound.

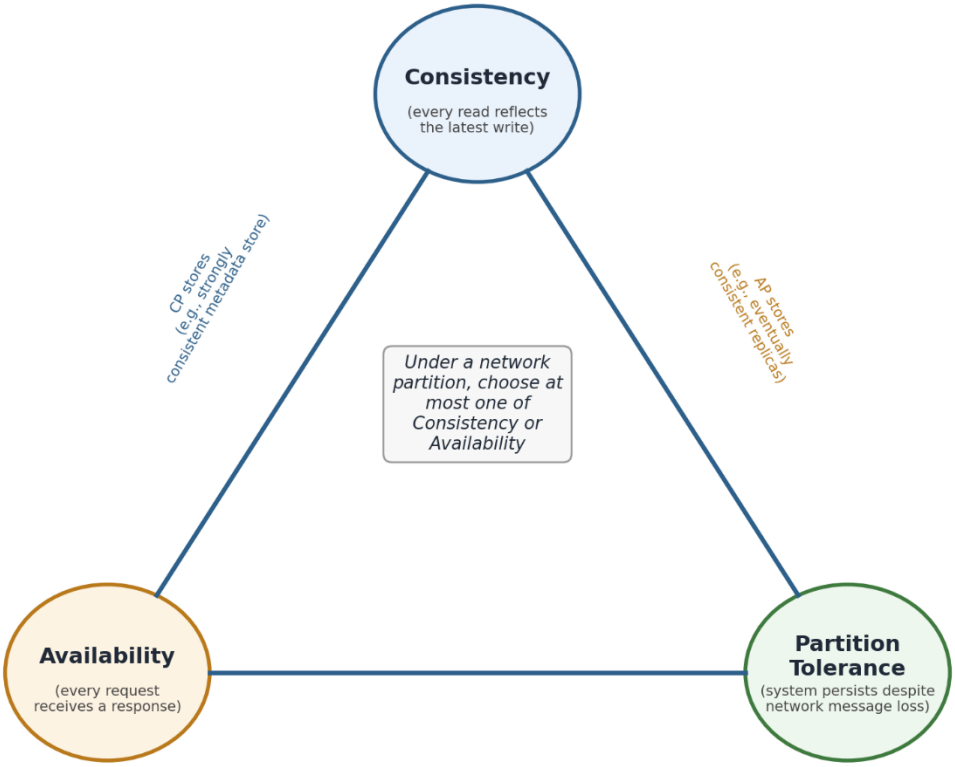

Figure 2. The CAP Trade-Off as It Applies to Individual Polyglot Data Stores
Source: Adapted from the impossibility result formalized in [11] and the eventual-consistency discussion in [10].

The first tension that emerges from the reviewed sources is the tension created between schema flexibility and schema rigidity. The NoSQL literature supports schema-on-read stores in part because they allow for more rapid and uncoordinated evolution [3], [13]. However, the empirical schema-evolution literature suggests that schemas in both relational and NoSQL models tend to become rigid after deployment, and that, due to their implicit nature, NoSQL schemas are more difficult to observe and control accurately in practice [8], [9]. This tension increases when reading the literature on state transfer carried by events: since an event schema is by its nature used by services that are not necessarily designed by the producer, there's no lack of schema contract, only the point at which incompatibility is found is shifted from the design phase to the runtime phase. This problem is addressed in the literature on schema evolution by defining a compatibility mode (backward, forward, or full) for the producer's proposed schema version, performing a compatibility check against that mode, registering the version, and publishing it; consumers will perform upcasting or downcasting to reconcile the version they read with the version they were built against [16], [17].

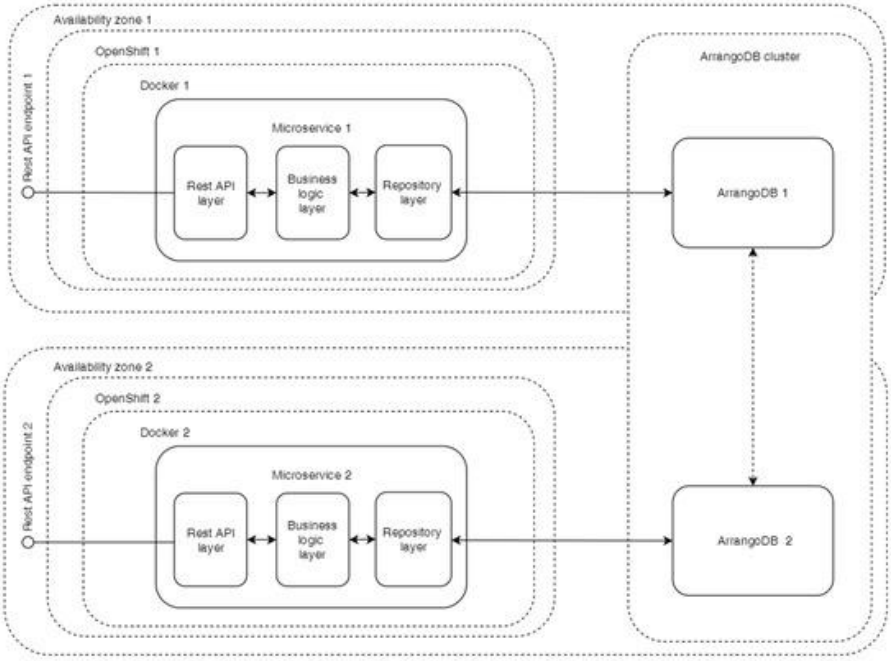

Figure 3. An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database (MDPI,2022)
Source: Synthesized from the versioning and migration strategies described in [17] and [16].

Table 3. Schema-Compatibility Modes and Their Implications for Event-Carried State Transfer

Compatibility mode	Guarantee provided	Implication for consumers
Backward	New schema can read data written with the previous schema	Consumers may upgrade after producers
Forward	Previous schema can read data written with the new schema	Consumers may upgrade before producers
Full	Both backward and forward guarantees hold simultaneously	Producers and consumers may upgrade independently, in any order
None	No compatibility contract enforced	Every schema change requires coordinated, simultaneous upgrade of producers and consumers

Source: Compatibility-mode definitions synthesized from the schema-evolution and migration literature reviewed in Section 2.2 [16], [17].

The second tension is between compensation-based transaction management and schema evolution. The compensating-transaction formalism, restated in Equation (3), is fundamental to the saga literature and it makes the assumption that transaction Tk can be undone by transaction (compensation) Ck, in spite of whatever other transactions occurred in the intervening time since either Tk committed or Ck executed. A compensation, however, in a polyglot, event-carried setting might have to be performed against a different service whose local schema has since changed and against event data that was created in a previous schema version. This saga-enhancement literature recognizes related operational challenges, such as compensations failing, or sagas interweaving over shared resources, but does not consider a failure in the schema drift between the original transaction and its later compensation itself as a failure mode [21], [22]. This framework is synthesized here, which proposes that compensations be treated as consumers like the events they are meant to compensate, with the same schema-registry guarantees provided for ordinary

event consumers as are provided for event consumers as shown in Figure 4.

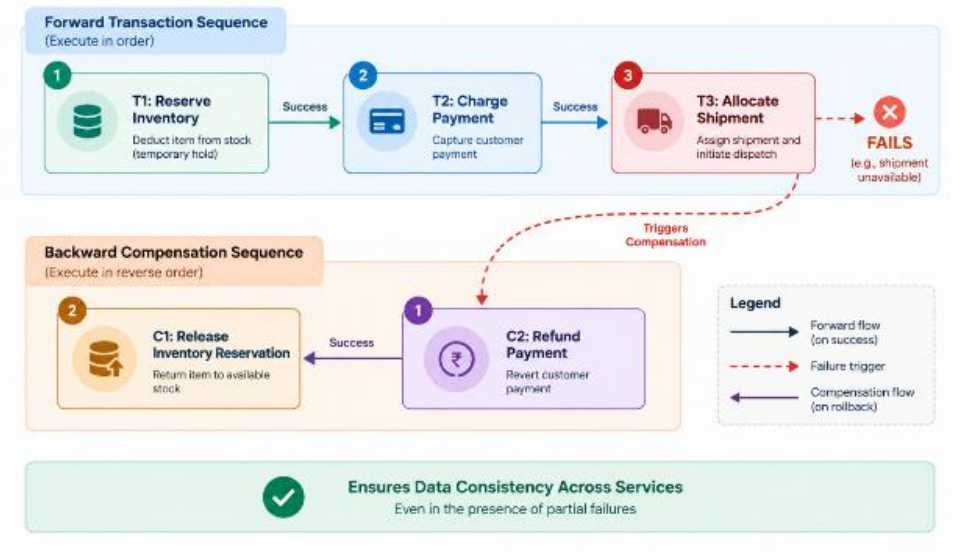

Figure 4. Saga Compensation Flow for a Cross-Service Transaction
Source: Synthesized from the original saga formalism [19] and its microservices application [20].

The third is the issue of the role of container orchestration. The orchestration literature views scheduling, scaling and service discovery as orthogonal to the data layer, and focuses on consistency and schema issues only indirectly in the choice of placement of services and network policies [5]. As shown in the synthesis, there is no widespread adoption of gating service deployment of a new version on a successful schema-compatibility check against the shared schema-registry, described as an open direction in the discussion in Section 5, in the literature of orchestration up to 2023 reviewed here.

The four figures and three tables in this section all point to the fact that schema governance, event-carried state transfer and saga-based transaction management are complementary pieces of a single consistency-management approach and not standalone architectural decisions. A schema change or state change in one service becomes visible across services according to consistency posture, which is chosen per store based on the CAP trade-off in Figure 2, and the visibility is schema-compatibility checked when it does (shown in Figure 3); finally, according to the synthesis, when a cross-service operation cannot be completed, the system recovers using saga-based compensation (shown in Figure 4), which must itself be schema-compatibility checked.

5. CONCLUSION

This review merged four related streams: microservice decomposition and polyglot persistence, empirical schema-evolution studies in NoSQL and polyglot stores, event-based integration via event-carried state transfer, and distributed transaction management via the saga pattern, and presented them in a single conceptual framework for schema-evolution-safe polyglot persistence in containerized microservices. The synthesis reveals that it is not sufficient for schema flexibility at the individual data-store level to solve the coordination issue caused by schema evolution when state is shared across services via events; that schema-on-read and schema-in-the-registry are complementary solutions that are independent of the underlying stores, as illustrated in the second and third figures in this paper. It also suggests that the way the reviewed literature currently views compensation-based transaction management

as the typical approach to the loss of cross-store atomicity should be considered a compatibility-aware consumer of the event log, and not as a process guaranteed to work with data whose schema is fixed. Since this review only draws from sources published up to 2023, its conclusions can only be interpreted as a summary of the research published until then; the open questions raised here, especially the maturity of tools to ensure consistency across stores and the maturity of container-orchestration platforms to incorporate schema-lifecycle policy, are presented as suggestions for empirical research rather than as final conclusions.

REFERENCES

[1] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," *MartinFowler. com*, vol. 25, no. 14–26, p. 12, 2014, [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.," 2021.

[3] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013. doi: 10.5555/2381014.

[4] N. Dragoni *et al.*, "Microservices: yesterday, today, and tomorrow," *Present ulterior Softw. Eng.*, pp. 195–216, 2017, doi: 10.1007/978-3-319-67425-4_12.

[5] E. Casalicchio, "Container orchestration: A survey," *Syst. Model. Methodol. tools*, pp. 221–235, 2018, doi: 10.1007/978-3-319-92378-9_14.

[6] C. ASHWIN, "Exploring event-driven architecture in microservices-patterns, pitfalls and best practices," *Int. J.*, vol. 4, no. 1, pp. 229–249, 2021, doi: 10.30574/ijsra.2021.4.1.0166.

[7] M. Kleppmann, *Designing data-intensive applications*. O'Reilly Beijing, 2017.

[8] P. Vassiliadis, A. V Zarras, and I. Skoulis, "Gravitating to rigidity: Patterns of schema evolution–and its absence–in the lives of tables," *Inf. Syst.*, vol. 63, pp. 24–46, 2017, doi: 10.1016/j.is.2016.06.010.

[9] S. Scherzinger and S. Sidortschuck, "An empirical study on the design and evolution of NoSQL database schemas," in *International Conference on Conceptual Modeling*, Springer, 2020, pp. 441–455. doi: 10.1007/978-3-030-62522-1_33.

[10] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, 2013, doi: 10.1145/2447976.2447992.

[11] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002, doi: 10.1145/564585.564601.

[12] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *J. Syst. Softw.*, vol. 150, pp. 77–97, 2019, doi: 10.1016/j.jss.2019.01.001.

[13] A. Davoudian, L. Chen, and M. Liu, "A survey on NoSQL stores," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 1–43, 2018, doi: 10.1145/3158661.

[14] J. Kazanavičius, D. Mažeika, and D. Kalibatienė, "An approach to migrate a monolith database into multi-model polyglot persistence based on microservice architecture: A case study for mainframe database," *Appl. Sci.*, vol. 12, no. 12, p. 6189, 2022, doi: 10.3390/app12126189.

[15] H. Singhal, A. Saxena, N. Mittal, C. Dabas, and P. Kaur, "PolyGlot persistence for microservices-based applications," *Int. J. Inf. Technol. Syst. Approach*, vol. 14, no. 1, pp. 17–32, 2021, doi: 10.4018/IJITSA.2021010102.

[16] M. Klettke, U. Störl, M. Shenavai, and S. Scherzinger, "NoSQL schema evolution and big data migration at scale," in *2016 IEEE International Conference on Big Data (Big Data)*, IEEE, 2016, pp. 2764–2774. doi: 10.1109/BigData.2016.7840924.

[17] A. Hillenbrand, U. Störl, M. Levchenko, S. Nabiyev, and M. Klettke, "Towards self-adapting data migration in the context of schema evolution in NoSQL databases," in *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, IEEE, 2020, pp. 133–138. doi: 10.1109/ICDEW49219.2020.00013.

[18] A. Chavan, "Eventual consistency vs. strong consistency: Making the right choice in microservices," *Int. J. Softw. Appl.*, vol. 14, no. 3, pp. 45–56, 2021, doi: 10.30574/ijsra.2021.1.2.0036.

[19] H. Garcia-Molina and K. Salem, "Sagas," *ACM Sigmod Rec.*, vol. 16, no. 3, pp. 249–259, 1987, doi: 10.1145/38713.38742.

[20] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[21] E. Daraghmi, C.-P. Zhang, and S.-M. Yuan, "Enhancing saga pattern for distributed transactions within a microservices architecture," *Appl. Sci.*, vol. 12, no. 12, p. 6242, 2022, doi: 10.3390/app12126242.

[22] K. Reza and N. Rahman, "Explication and Extension of Saga and Microservice Patterns to Enable Resilient Distributed Transaction," in *Inventive Communication and Computational Technologies: Proceedings of ICICCT 2022*, Springer, 2022, pp. 209–220. doi: 10.1007/978-981-19-4960-9_18.

[23] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 22–32, 2017, doi: 10.1109/MCC.2017.4250931.