Universal Library of Innovative Research and Studies Research Article

ISSN: 3065-0003

Open Access | PP: 20-25

DOI: https://doi.org/10.70315/uloap.ulirs.2023.004

Method for Supporting Consistent Data State in Microservice Systems

Kostadin Almishev

Abstract

Within the framework of the article, an improved approach to maintaining a consistent data state is considered, based on the Saga pattern and implemented through centralized orchestration in combination with persistent storage of the operation context in a NoSQL database. The objective of the work is the design and experimental validation of execution algorithms, compensating rollback, and subsequent recovery of operations, focused on reducing response time and eliminating a single point of failure. The methodological basis includes functional decomposition of processes using the IDEF0 notation and conducting load testing of the developed software complex implemented in the Kotlin language and using Apache Kafka. The experimental part was performed on banking data with a volume of 1.5 TB; the obtained results show that the proposed method provides an increase in the stable load threshold by 8% and a reduction in the number of transactions in the database by 18% relative to the two-phase commit protocol (2PC). The final provisions confirm the expediency of applying the model of eventual consistency (BASE, Basically Available, Soft state, and Eventually consistent theorem) under the conditions of high-load financial systems, where a balance between performance and preservation of strict guarantees of data integrity is required. The information reflected in the article will be of practical significance for system architects and developers of distributed platforms interested in optimizing performance while maintaining data correctness.

Keywords: *Microservice, Distributed System, Data Consistency, Data Integrity, Transaction, Saga Pattern, Orchestration, Two-Phase Commit, Fault Tolerance, Cloud Computing.*

INTRODUCTION

The modern software development industry is undergoing a phase of intensive modernization, the key manifestation of which is a large-scale transition from monolithic applications to microservice architectures (MSA) [1]. In the scientific and applied domains, this trend increases the significance of the discipline of data management: under conditions of growing complexity of the IT landscape, it is precisely the quality of data management that becomes an independent factor of competitiveness [2, 3].

Despite all well-known advantages of MSA—independence of service life cycles, horizontal scalability, and technological autonomy—the central problem remains ensuring a consistent data state when executing distributed transactions. In an architectural model where each service owns its own database (the database-per-service pattern), classical ACID (Atomicity, Consistency, Isolation and Durability) guarantees cease to be a property of the system as a whole and are reduced to the local level of individual storages. A scientific-practical gap arises: there are no standard, industrially applicable solutions capable of simultaneously maintaining high availability (in the logic of the CAP (Consistency, Availability, Partition tolerance) and avoiding significant performance degradation typical of traditional coordination protocols such as two-phase commit (2PC). Additional complexity is introduced by the fact that widespread implementations of the Saga pattern often prove to be either excessively labor-intensive in maintenance and debugging, or susceptible to the risk of partial inconsistency in the event of coordinator failures and message delivery violations [4].

The formulated **objective** of the study is to develop and experimentally evaluate a method for supporting a consistent data state in microservice systems based on an orchestrated Saga, oriented toward increasing fault tolerance and minimizing transactional overhead. As a methodological basis, the thesis is adopted on the necessity of explicit management of the context of a distributed operation: without a formalized state and strict invariants of transitions, it is impossible to ensure predictability of results under network delays, message retries, and partial component failures.

The scientific novelty of the proposed approach lies in the substantiation of a context-driven orchestration algorithm that introduces a semantic red line (pivot point). This boundary delineates the phase of reversibility—managed via compensating local transactions—from the phase of guaranteed completion (forward recovery). By replacing monolithic distributed transactions with a sequence of local autonomous transactions, this method ensures consistency across heterogeneous systems (including external ones), minimizes global locking, and optimizes Database Management System (DBMS) resource utilization .

Additional substantiation of the relevance of the approach follows from the reliability characteristics of distributed systems: the presence of retries, non-strict message delivery,

and temporary unavailability of individual services turns idempotency of operations and determinism of state transitions into mandatory properties of the transactional contour. Practically significant is the formation of a resilient scheme for deduplication of commands and events, as well as the use of store + message bus coordination techniques, including the transactional outbox/inbox patterns, ensuring correct publication of domain events without divergence between a write to the database and sending a message. In the context of an orchestrated Saga, this mandates strict step sequencing, context versioning, and a regulated compensation mechanism. Ensuring the measurability and control of side effects is critical, particularly when interacting with external systems involving conditionally reversible operations (e.g., payments prior to settlement) or inherently irreversible actions (e.g., notifications).

The author's hypothesis is formulated as follows: application of the proposed method with persistent storage of the context of a distributed operation in MongoDB is capable of providing an increase in system throughput by 8% and a reduction in the maximum execution time of operations by 25% compared to the 2PC protocol. This assumption is based on the expected reduction of coordination overhead and shifting part of reconciliations into a managed state context. Critically, it involves moving from distributed locking to granular application-level locking (semantic locks). While this retains necessary constraints on specific resources, it avoids the systemic long-lived global locks typical of 2PC, which are especially pronounced under network degradation and partial failures, causing increased tail latencies and reducing overall service stability.

MATERIALS AND METHODS

The methodological basis of the study was formed on the basis of a comprehensive analysis of practices and theoretical models of designing distributed systems. A systematic literature review was applied: the results of other studies in recent years devoted to data management in MSA [6] and works on distributed transaction patterns [7] were considered. For rigorous formalization of the subject domain, functional decomposition in the IDEF0 notation was used, which made it possible to clarify the problem statement and detail the contours of execution, compensating rollback, and recovery of operations. Additionally, a comparison of the conceptual paradigms ACID and BASE (Basically Available, Soft State, Eventual Consistency) was performed in the context of high-load financial systems, where response predictability and resilience to partial failures are required [8].

Experimental verification was implemented through the development of a library in the Kotlin language and the construction of a test bench including Spring Cloud Stream, Apache Kafka, and the NoSQL storage MongoDB used to store context of operations [11]. The empirical part is based on the author's computational and explanatory note of the final qualifying work, in which the algorithms for ensuring data integrity by introducing an additional operation context are presented in detail. The specified context includes the process identifier (processId), the creation timestamp (createTime), the version, as well as status states describing the life cycle of a distributed operation (STARTED, COMMITED, COMPLETED, NEED_ROLLBACK, ROLLED_BACK).

RESULTS AND DISCUSSION

In the course of the study, a method was formalized and implemented in software that extends the standard Saga pattern by introducing mechanisms for deterministic management of stalled distributed operations, as well as ensuring idempotency of suboperations at the execution protocol level. In contrast to typical implementations, where resilience is achieved predominantly through command retries and compensating transactions, the proposed approach defines a formal model of the context life cycle with fixed invariants and unambiguous rules for transitions between states. This makes it possible to reproducibly restore execution under partial failures, network disruptions, and repeated message delivery, eliminating uncontrolled multiple activation of steps and accumulating inconsistency.

Architecturally, the method introduces a separation of microservices into two classes: facade orchestrator components and data access services that isolate interaction with the database. The orchestrator concentrates the business process logic and is responsible for managing the sequence of steps, while each step is recorded in a specialized storage of distributed operation contexts. The recording includes the transaction context identifier, the step number, the state version, the execution result, indicators of the need for compensation, as well as reprocessing metadata (for example, an attempt counter and timestamps). Decomposition into orchestrator — data access service reduces the coupling of domain logic with the storage infrastructure and simultaneously creates a strictly defined boundary of responsibility: data access services perform locally atomic operations with their own ACID guarantees, whereas consistency at the business process level is ensured by orchestration and the context state.

A key element of the algorithm is the transition across a semantic red line—a suboperation after the successful completion of which rolling back the business process becomes inadmissible or economically unacceptable. After passing this threshold, the transaction is transferred into the guaranteed completion mode: the further behavior of the system is determined by the principle of forward recovery, that is, the mandatory completion of the remaining steps to a terminal state under any permissible failures. In formal terms, the pivot point marks the boundary at which the recovery strategy changes: before it, the operation can still be rolled back; after it, the system switches to a continuation strategy, under which it must achieve a stable final state through retries, context reconstruction, and repeated command delivery, without undoing already accepted irreversible decisions.

An additional strengthening of the industrial applicability of the method is the explicit inclusion of idempotency in the contract of suboperations. For each suboperation, a unique execution key is recorded (usually a composition of the context identifier and the step number), by which the data access service is able to detect a retry and return the previously recorded result without repeated impact on the state. Thus, resilience is achieved to at least once message delivery and to retries initiated by orchestrator timeouts. Management of stalled operations is ensured by the fact that the context stores not only the fact of step execution, but also its status in terms of progress (initiated, in progress, completed, requires intervention), which makes it possible to safely resume the process after an orchestrator restart or under degradation of individual services without the risk of double application of side effects. Below in Figure 1-3 is demonstrated the functional scheme of the method in the IDEF0 notation.

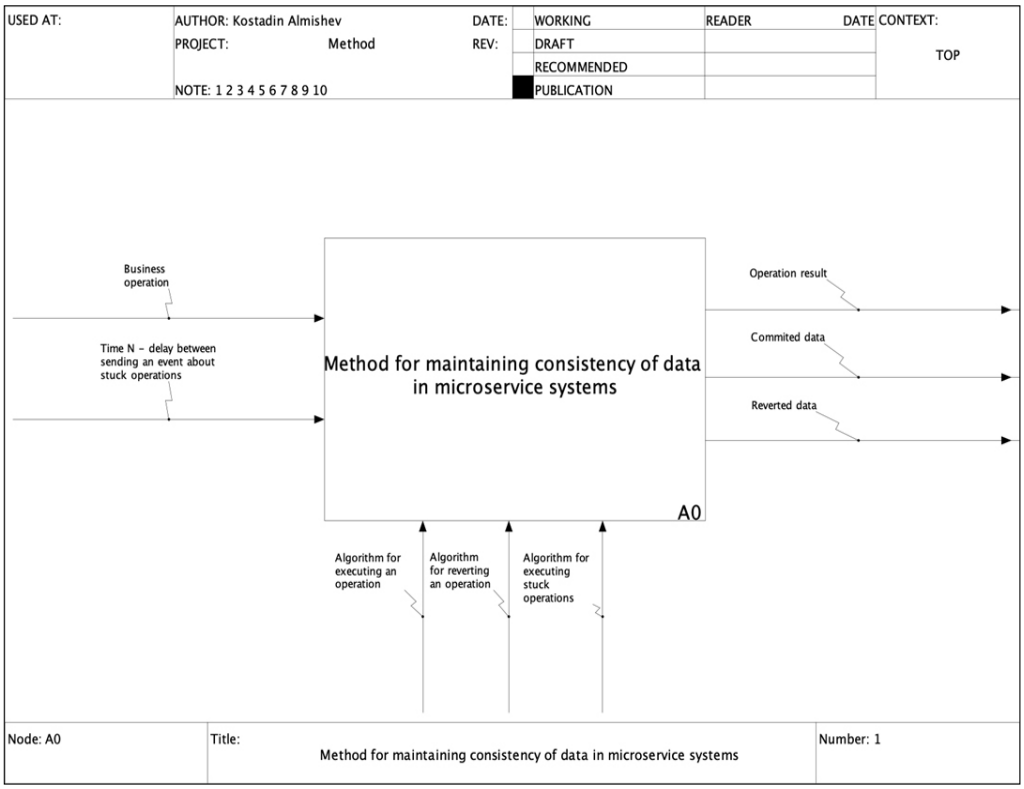

Fig.1. Method for maintaining consistency of data in microservice systems (author's data).

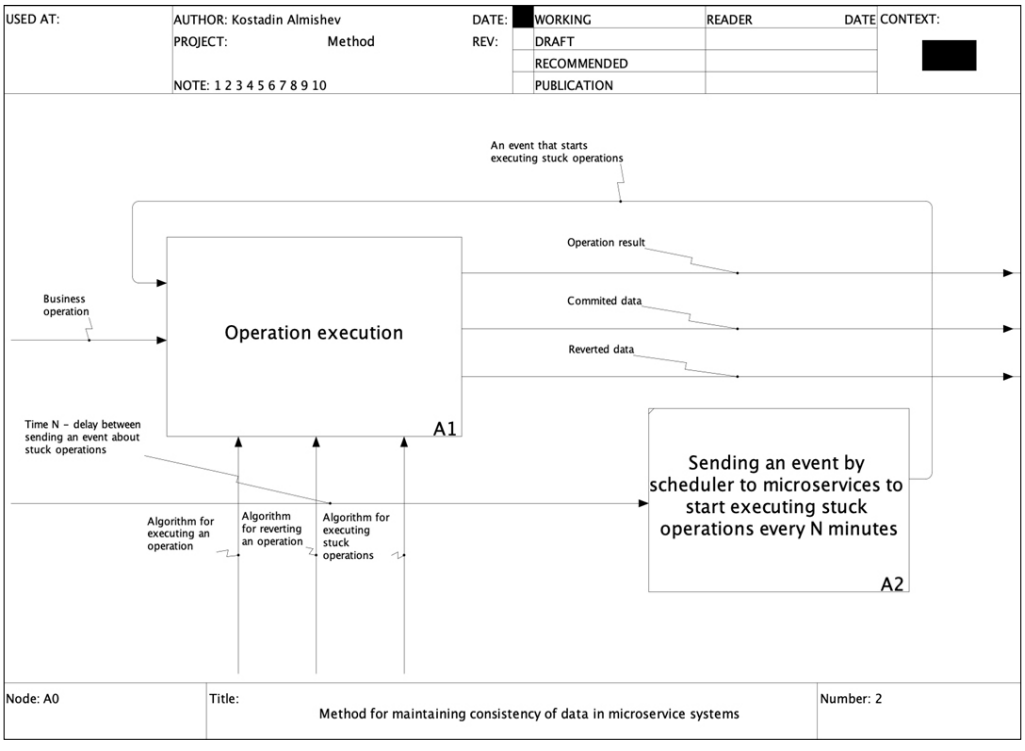

Fig.2. Method for maintaining consistency of data in microservice systems (author's data).

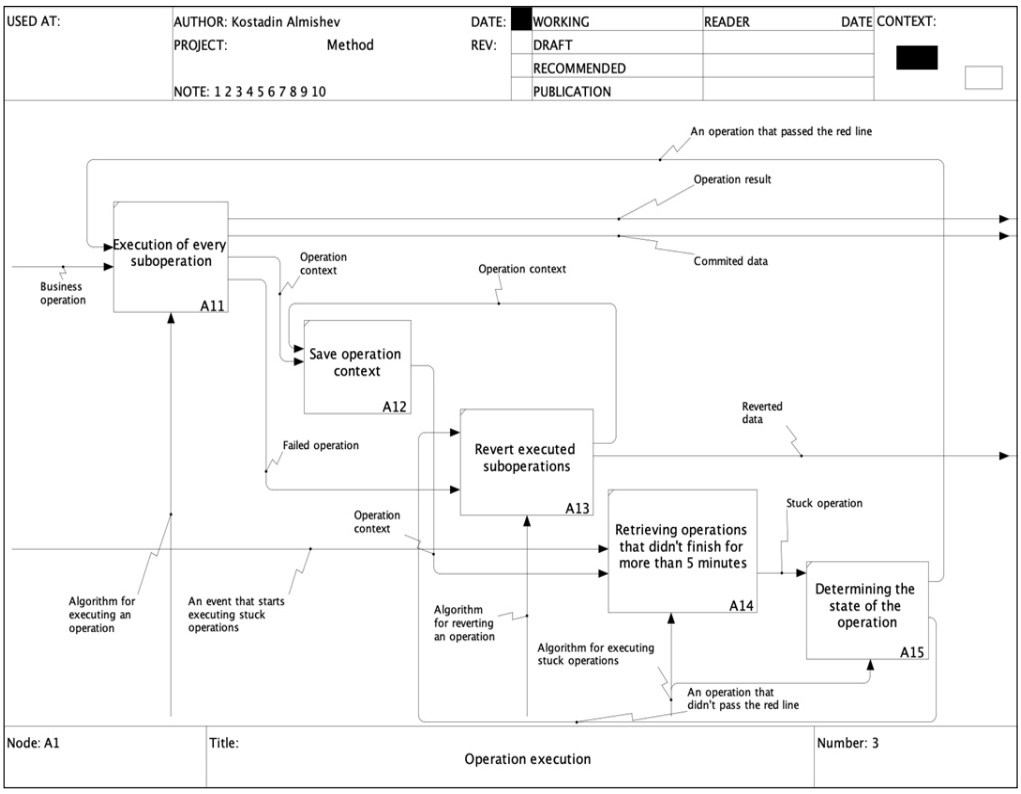

Fig.3. Operation execution (author's data).

The experimental evaluation of throughput was performed at load levels of 200, 300, 325, and 350 operations per second. The criterion for the absence of degradation was defined as maintaining a correct request processing mode without the occurrence of server errors of the 500/503 classes, without exceeding the maximum response waiting time (timeout), interpreted as exceeding permissible latency values under the specified intensity of incoming operations, and while preserving the expected conversion rate. In the observation window, conversion was calculated as follows:

$$Conversion = \frac{N_{successful}}{N_{created}} \quad (1)$$

where: $N_{successful}$ is the number of successful operations and $N_{created}$ is the number of created operations.The obtained quantitative results were aggregated and are presented in Table 1.

Table 1. Degradation of functionality depending on the number of operations (author's data).

Operations per second	Two-phase commit (2PC)	Improved Saga
200	-	-
300	-	-
325	+	-
350	+	+

The analytical interpretation of the experimental results indicates that for the 2PC protocol, the boundary of stable operation is reached at an intensity of 300 ops/sec, after which signs of degradation are recorded. In turn, the improved Saga implementation maintains a correct operating mode up to 325 ops/sec, which is equivalent to an increase in throughput by 8.3%. The achieved effect is explained by the fact that within the proposed approach, the need to hold locks on shared resources throughout the entire distributed transaction is eliminated, as a result of which contention for locks and the probability of a cascading increase in delays are reduced.

A comparison of the transactional load on the DBMS demonstrates that reducing the number of database accesses frees computational resources and lowers the overhead of transaction management, redistributing performance in favor of processing application load. The quantitative indicators of the number of transactions obtained during the experiments are summarized and presented in Table 2.

Table 2. Average number of transactions in the database per second depending on the load (author's data).

Number of operations per second	Two-phase commit	Improved Saga
200	6 794	4 320
300	10 623	7 231
325	-	9 833
Average per operation	35	25

The averaged indicator of transactions per one business operation in the proposed method is 18% lower compared to 2PC. This effect is due to by the elimination of coordination overheads typical of 2PC: the voting phase is absent, and the volume of synchronous writes to transaction logs on the DBMS side is also reduced, since interservice consistency is achieved not through global commitment of a distributed

transaction, but through management of the execution context. Heavyweight operations of logging and holding resources are replaced by lighter state records in the NoSQL storage of contexts, which reduces pressure on input-output subsystems and decreases the probability of degradation under peak loads, especially in scenarios with high contention for locks.

In the performance contour, the key parameter for banking APIs is not only the average latency, but primarily the maximum operation execution time, assessed via the latency of the 90% quantile (P90). This indicator reflects the tail of the latency distribution and characterizes system behavior under unfavorable, but statistically significant conditions: under transaction contention, network variations, partial failures, and repeated message delivery. In architectures oriented toward strict SLAs, it is precisely tail latencies that determine the perceived quality of service, since even rare latency spikes lead to timeouts on the consumer side, avalanche retries, and secondary overloads. In this regard, reducing coordination overhead in favor of an asynchronously managed context forms prerequisites for stabilizing P90 by decreasing the number of blocking synchronous phases and reducing the transactional footprint in the DBMS.

Figure 4 will reflect the dependence of execution time on request intensity.

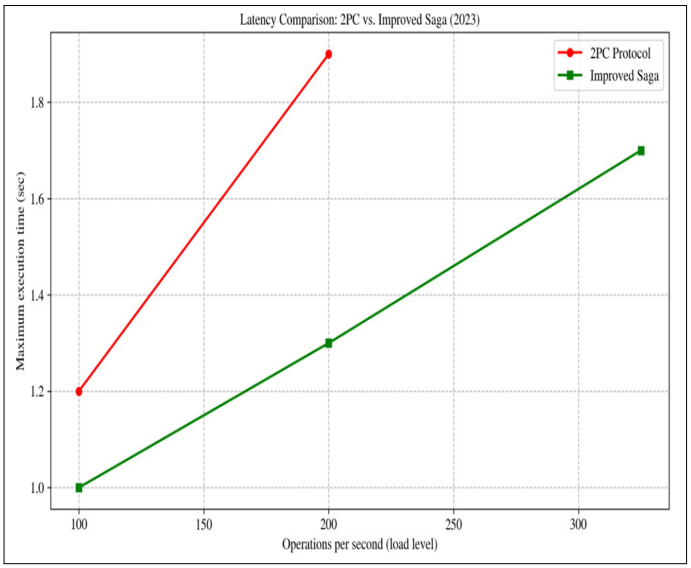

Fig. 4. Dependence of execution time on query intensity (author's data).

The average execution time of an operation within the proposed method is 1.2 sec, which is 25% less than when using 2PC (1.6 sec). At the same time, the average time for saving and reading the context in MongoDB is 1.6 ms, that is, less than 3% of the total transaction duration. This proportion indicates a statistically negligible contribution of the orchestration contour to the overall latency and confirms that the gain is achieved mainly due to the reduction of synchronous coordination phases and the elimination of long-lived locks characteristic of classical distributed commit. In terms of operational reliability, it is also important that the low cost of context operations makes it possible to apply more frequent progress fixation without noticeable deterioration of latency, increasing resilience to failures and reducing the probability of re-executing long chains of steps.

Despite the observed superiority in speed, the method retains a number of fundamental limitations stemming from the very nature of sagas and the foregoing from global transactional isolation. The key risk is associated with the absence of isolation (I in ACID) at the level of the entire business process until the completion of the entire Saga. Under such conditions, concurrency access anomalies may occur, including dirty reads, reading intermediate states, and effects of inconsistent data visibility, when some services have already committed local changes while others are still in the process of execution. This makes traditional expectations of as in a monolith insufficient and requires the introduction of application semantic locks and rules of consistent visibility implemented at the domain model level. In practice, patterns based on entity statuses (for example, pending/confirmed/failed), prohibition of reading/using objects in an intermediate state, as well as the introduction of domain invariants that prevent decision making based on incomplete data are applicable.

An additional barrier is the increase in design complexity due to the need for explicit construction of compensating transactions for each step. Compensations are rarely a true inverse operation in the mathematical sense: in the presence of external effects, fees, exchange rate differences, and regulatory accounting constraints, compensation often takes the form of a separate business operation with its own constraints and audit. Errors in compensating design lead not so much to technical failures as to violations of business invariants and the emergence of financially significant discrepancies. In banking systems, such risks are mitigated by audit, tracing, and API versioning regulations, as well as by standardization of the operation life cycle: the reasons for state transitions, correlation identifiers, timestamps, contract versions, and re-execution policies are recorded. A significant role is also played by the formalization of conditions of irreversibility (pivot point): the more correctly the semantic boundary of guaranteed completion is defined, the lower the probability of transitioning into a state requiring labor-intensive manual settlement [9, 10].

The limitations also include high sensitivity of implementation quality to the discipline of idempotency and deduplication. With incorrect handling of command retries, duplication of side effects is possible (for example, double debit or repeated enqueueing), which requires strict idempotency keys, consistent storage of step results, and formal contracts for repeated invocation. Finally, the load on observability increases: end-to-end metrics for Saga states, correlation identifiers, as well as mechanisms for automatic pickup of stalled contexts are required, since otherwise operational advantages may be partially offset by increased time for diagnosis and recovery [6].

CONCLUSION

Within the framework of the conducted study, a method for maintaining a consistent data state in microservice systems, oriented toward operation under high load, was substantiated and experimentally confirmed. The performed analytical elaboration of the subject domain made it possible to fix the key limitations of the 2PC protocol in MSA environments, primarily associated with long-term holding of locks on resources and increased vulnerability of the coordination component, which collectively reduces stability and degrades response characteristics. On the basis of the identified problems, an approach was formed and implemented in which the orchestration algorithms of distributed operations are supplemented by a semantically defined boundary (red line) that transfers the transaction into a guaranteed completion state and thereby eliminates scenarios of data loss under failures through deterministic management of the operation life cycle.

Experimental verification showed a statistically significant advantage of the developed solution across the set of target metrics: the throughput increase amounted to 8%, the transactional load on the DBMS decreased by 18%, and the maximum duration of operation execution was reduced by 25%. The practical value of the results is confirmed by the creation of a Kotlin library providing the capability to scale financial applications without compromises in preserving data integrity. The formulated hypothesis was confirmed based on the results of load testing. A promising direction for further development is associated with extending the proposed methodology through predictive identification and management of stalled operations, relying on intelligent monitoring contours of the AIOps class.

REFERENCES

1. Flexera. (2023). Flexera 2023 State of the Cloud Report (Press release). Retrieved from: https://www.flexera.com/about-us/press-center/flexera-2023-state-of-the-cloud-report (date accessed: July 3, 2023).
2. Canalys. (2023). Worldwide cloud service spend to grow by 23% in 2023 (Press release). Retrieved from: https://canalys-prod-public.s3.eu-west-1.amazonaws.com/static/press_release/2023/1725098181Worldwide-Cloud-Market-Q4-2022.pdf (date accessed: July 11, 2023).
3. Razzaq, A. (2020). A systematic review on software architectures for IoT systems and future direction to the adoption of microservices architecture. SN Computer Science, 1(6), 350.
4. Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. A. (2021). Understanding and addressing quality attributes of microservices architecture: A systematic literature review. Information and Software Technology, 131, 106449. https://doi.org/10.1016/j.infsof.2020.106449
5. Ioannidis, P. I. (2023). Distributed transactions using the SAGA pattern (Master's thesis, National and Kapodistrian University of Athens). Pergamos. Retrieved from: https://pergamos.lib.uoa.gr/uoa/dl/object/3313109/file.pdf (date accessed: August 2, 2023).
6. Stefenko, M., & Zdun, U. (2019). The saga pattern in a reactive microservices environment. In Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019) (pp. 483–490). SCITEPRESS. https://doi.org/10.5220/0007918704830490
7. Bhole, A. (2023). Challenges and strategies for consolidating data across microservices. International Journal of Innovative Research and Creative Technology, 9(1), 1–10. https://doi.org/10.5281/zenodo.14598748
8. Maximize Market Research. (n.d.). Microservices Architecture Market: Global Industry Analysis and Forecast (2023–2029). Retrieved from: https://www.maximizemarketresearch.com/market-report/microservices-architecture-market/17318/ (date accessed: August 14, 2023).
9. Daraghmi, E., Zhang, C.-P., & Yuan, S.-M. (2022). Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. Applied Sciences, 12(12), 6242. https://doi.org/10.3390/app12126242
10. Spring. (2023). Spring Framework 6.0.6 Reference Documentation. Retrieved from: https://docs.spring.io/spring-framework/docs/6.0.6/reference/pdf/spring-framework-reference.pdf (date accessed: September 5, 2023).
11. Apache Software Foundation. (2023). Apache Kafka 3.4.0 Documentation (site docs). Retrieved from: https://archive.apache.org/dist/kafka/3.4.0/kafka_2.12-3.4.0-site-docs.tgz (date accessed: September 18, 2023).

Citation: Kostadin Almishev, "Method for Supporting Consistent Data State in Microservice Systems", Universal Library of Innovative Research and Studies, 2023; 20-25. DOI: https://doi.org/10.70315/uloap.ulirs.2023.004.