УДК 004.272:004.75 **DOI:** 10.31673/2412-9070.2025.061213

A. KOLODIUK[1], Ph.D. student;
ORCID: 0009-0001-1724-7531
O. VOLOSCHUK[2], Ph.D., associate professor,
ORCID: 0000-0002-5912-4126
[1]State University of Information and Communication Technologies, Kyiv
[2]V. N. Karazin Kharkiv National University, 4 Freedom Square, Kharkiv

PER-ORDER ORDERED PARALLELISM METHOD IN MICROSERVICE EVENT DELIVERY SYSTEMS

This paper presents a per-order ordered parallelism method for event delivery in microservice-based architectures, designed to ensure strict execution order within each logical workflow while sustaining high throughput and fault tolerance. The proposed approach introduces an application-level ordering layer built atop the RabbitMQ message broker, avoiding any modification of broker internals. The method relies on per-event sequence identifiers (X-Sequence-ID), dynamic per-key queue instantiation, and a gap-replay mechanism for recovering missing states, thereby providing deterministic processing and reliable restoration of system state even under message loss or redelivery.

The scientific contribution consists in the development of a formal execution model that integrates at-least-once delivery semantics, idempotent event handling, controlled redelivery, and bounded buffering. An analytical model is constructed to quantify the influence of parallelism on system latency, throughput, and stability, taking into account probabilistic message loss, queue load factors, resource constraints, and fallback HTTP channels. This model enables formal evaluation of the system overhead ε and predictive assessment of behavior during broker degradation, partial failures, or system updates, offering a mathematically substantiated basis for tuning concurrency and recovery mechanisms.

Experimental evaluation within the AutoGivex microservice ecosystem demonstrates reduced processing latency, increased stability, and improved scalability compared to traditional broker-level ordering strategies. The results confirm that the proposed method provides a robust foundation for building resilient event-driven distributed systems that require strict per-order consistency, deterministic behavior, and adaptive throughput optimization. The findings can be applied to a wide range of domains—financial, logistics, enterprise, and real-time platforms—where correctness and reliability of event sequencing are essential.

Keywords: ordered parallelism; microservices; event delivery; RabbitMQ; X-Sequence-ID; idempotency; gap-replay; concurrency model; fault tolerance; analytical model; AutoGivex information technology.

Introduction

In modern distributed information systems built on the principles of microservice architecture and event-driven interaction, a key factor in ensuring reliability and performance is the consistency of event processing order. The growing scale of service integration, the distributed nature of computational resources, and the increasing number of concurrent user requests create the need to maintain a high level of parallelism while preserving a strict order of operations within each logical context – such as an order, transaction, or user session.

In most existing message brokers, including RabbitMQ, Kafka, and NATS, ordering mechanisms are implemented at the level of queues or partitions and do not guarantee consistent ordering for individual keys under high-load conditions. This leads to the problem of event reordering, where asynchronous delivery results in violations of business process logic. In event delivery systems where processing steps must occur in a strictly defined sequence (for example, confirming payment before generating an invoice), such disruptions are critical.

The developed per-order ordered parallelism method becomes particularly relevant in ensuring local determinism of event processing while maintaining global system parallelism. Its essence lies in implementing application-level ordering on top of the standard RabbitMQ message broker by using sequence identifiers (X-Sequence-ID), dynamic creation of per-key internal queues, and a gap-replay mechanism for recovering missing states. This approach allows the system to maintain the simultaneous execution of a large number of event streams without violating logical consistency within individual keys.

The relevance of this development is further emphasized by the limitations of current ordering approaches (such as key-partitioning in Kafka or message grouping in AWS SQS), which include a fixed number of processing streams, difficulties in scaling under uneven load distribution, and the lack of built-in mechanisms for recovering order in cases of message loss or duplication. The proposed method eliminates these shortcomings through adaptive queue management and integrated sequence control at the business logic level, without interfering with the transport layer.

Contemporary literature on event delivery in microservice architectures focuses on two tightly coupled topics: ensuring correct message ordering and improving reliability/fault tolerance via patterns such as the transactional outbox, idempotent APIs, and replay mechanisms. Broker primitives (e.g., RabbitMQ) provide publisher confirms and consumer acknowledgements, but ordering guarantees are local to a queue or partition; this limitation motivates shifting some ordering logic to the application layer or combining broker features with outbox patterns [1–3]. The Transactional Outbox and its implementations (polling publisher, transaction-log tailing, CDC) are widely recommended as standard tools for at-least-once guarantees and for enabling event replay in case of missing messages [4, 6, 8]. Practitioner guides detail the trade-offs between immediate insert→publish vs. WAL/CDC-based approaches and stress the need for idempotency to make retries safe [5, 7, 9].

Idempotency and safe retries are emphasized in industry guidance: use of idempotency tokens, careful control of side effects, and retry strategies with exponential backoff and jitter are standard recommendations to avoid duplicate side effects during retries [15-17]. The body of tutorials and blog posts explains how to use publisher confirms, routing keys, and fallback channels in production and highlights operational challenges – hot keys, buffer sizing, and backpressure—when implementing per-key ordering at scale [5, 9, 10].

Recent academic research analyzes the limits of global ordering and proposes sequencer- or consensus-based approaches as well as practical heuristics; these works formalize trade-offs among latency, throughput, and fairness and show that in many real-world scenarios it is preferable to provide per-key ordering with replay mechanisms rather than expensive global total order via consensus [12-14]. Empirical studies on event management in microservices show developers face gaps in tooling for event auditing, ordering, and recovery, supporting the practical need for persisted state logs (JobStates) and built-in gap-replay as recovery mechanisms [12, 19]. Open-source replay tools and event-mesh documentation provide practical blueprints that can be adapted when implementing per-key ordered parallelism in production systems [11, 18, 20].

Taken together, the literature provides a solid theoretical and practical grounding for the per-order ordered-parallelism approach: combining X-Sequence-ID tagging, dynamic per-key queues, Transactional Outbox, publisher confirms, and gap-replay aligns with both established industry patterns and academic insights while enabling correctness under high concurrency without broker modification [1-20].

Research problem statement

The aim of this study is to develop a method and analytical model of per-order ordered parallelism for microservice-based event delivery systems, ensuring strict sequential consistency for each logical order while maintaining high throughput and resilience against broker or network failures. The research focuses on implementing an application-level ordering mechanism on top of the RabbitMQ message broker using X-Sequence-ID identifiers, a gap-replay recovery mechanism, and constructing an analytical model to evaluate the impact of parallelism on performance and latency under at-least-once delivery semantics.

To achieve the stated aim, the following objectives have been defined:

1. Analyze existing approaches to event ordering and delivery guarantees in microservice architectures, including broker-level mechanisms (RabbitMQ, Kafka, NATS) and application-level serialization strategies.

2. Develop a per-order ordered parallelism method based on event tagging with X-Sequence-ID, dynamic per-key queue creation, and a gap-replay mechanism for recovery of missing states.

3. Construct an analytical model describing the relationship between system parallelism, latency, and throughput while accounting for idempotency, redelivery, buffering, and backup HTTP transmitssion channels.

4. Perform a comparative performance analysis of the proposed method against conventional unordered or centralized queue-based systems to determine system overhead ε and optimal parallelism level.

5. Provide implementation guidelines for integrating the developed method into microservice architectures (specifically AutoGivex) to improve system stability, scalability, and correctness of event-driven processing in real-time environments.

Main results

To build a mathematical framework for describing and analysing the proposed method of ordered parallelism per-order, it is appropriate to combine theoretical approaches (queueing theory, stochastic processes, event/sequence models) with practical limitations of message brokers (RabbitMQ, Kafka) and patterns (event-sourcing / outbox) [21, 22].

Let the system operate on a set of keys (contexts) K.

An event $e$is defined as a triple (k, s, t), where $k \in K$ – the key (for example, an order identifier), $s \in \mathbb{N}$ – the local sequential number (X-Sequence-ID), $t \in \mathbb{R}_{\geq 0}$– the time the event arrives in the system.

We define the following notations:

– λ_k– the average rate of event arrivals for key k(events per second);

– μ_k– the average service rate for processing events with key k(events per second), which is the inverse of the average service time;

– $\rho_k = \lambda_k/\mu_k$– the utilization (load factor) of the local queue Q_k;

– $N_a(t)$– the number of active internal queues $Q_k$at time t;

– m– the number of available worker threads, each capable of serving one queue at a time;

– p_{loss}– the probability of event loss or non-persistence in the delivery channel (modeled as i.i.d. losses as a first approximation);

– T_{gap}– the waiting time for detecting a missing event in the sequence (gap timeout);

– R– the maximum number of allowed gap-replay attempts before declaring a failure;

– T_{attempt}– the average duration of a single gap-replay attempt (request – response).

Assumptions:

1. Events for each key k are numbered with monotonically increasing natural numbers s.

2. The producer assigns each event with key k a unique X-Sequence-ID $s$at the moment of its generation (using a local counter or a guaranteed sequence number generator for that key k).

3. The broker (RabbitMQ) guarantees FIFO delivery only within a single queue but does not ensure a global order across different queues or consumers. It may perform message redelivery (at-least-once) or allow message loss if message retention is disabled.

4. A gap-replay mechanism exists: upon request, the source system (event store or broker buffer) can replay any missing event with identifier $s$for key k, provided that the event has been retained. If the source no longer holds the event, the replay may fail.

These assumptions formalize the practical constraints of real-world message-driven systems and form the theoretical basis for subsequent proofs.

For each active key k, an internal queue Q_k is maintained along with a variable expected_seq[k] – the next expected sequence number. An incoming event (k, s, t), once received by the consumer, is placed into (Q_k).

Processing of elements from Q_k is permitted only when an event with *s=expected_seq[k]* is available. If the head of Q_k contains an event with *s'>expected_seq[k]*, the system waits up to T_{gap} for the missing event. After this timeout expires, the system initiates a gap-replay request for the missing sequence number *expected_seq[k]*.

Theorem 1 (Per-key ordering – Safety). Under the assumptions listed above, the per-key ordering algorithm guarantees that events for each key (k) are processed strictly in the increasing order of their sequence numbers (s).

Proof.

The system maintains an invariant: before processing queue Q_k, the value *expected_seq[k] = r* means that all events with numbers less than (r) have already been successfully processed. Processing is only permitted for events where ($s = r$). If the head of the queue contains an event with ($s' > r$), processing is postponed until the missing event arrives or is retrieved via gap-replay. Thus, no event with a higher number can be processed before one with a lower number, ensuring strict per-key ordering.

Theorem 2 (Liveness with Persistent Events). If the event source retains all published events for a sufficient period (retention time ≥ the maximum expected gap + replay time) and each replay atempt has a success probability ($p_s > 0$), then, given a finite number of attempts (R), the probability that any queue Q_k remains indefinitely blocked is zero – meaning the missing event will eventually be recovered with probability 1.

Proof (Sketch). Each gap situation triggers up to (R) independent replay attempts. The probability that all replay attempts fail equals $P_{fail} = (1 - p_s)^R$.

If the event source retains all data, ($p_s > 0$), and the number of attempts (R) is sufficiently large, the probability of permanent blockage tends toward zero. Hence, the system guarantees liveness under these conditions.

Theorem 2 highlights a critical design requirement: ensuring event persistence (e.g., through an outbox pattern, event-store, or broker message retention) is necessary to maintain liveness in the system.

Each per-key queue (Q_k) can be modeled approximately as an (M/M/1) queuing system: arrivals follow a Poisson process with rate (λ_k), and service times follow an exponential distribution with rate.

Lemma 3 (Stability Condition). If for every active key (k) the utilization $\rho_k = \frac{\lambda_k}{\mu_k} < 1$,and the total service capacity satisfies $\sum_{i=1}^{m} \mu_{k_i} > \sum_k \lambda_k$, then the system is stable – that is, expected queue lengths and waiting times are finite.

Proof (Sketch). For each Q_k, the condition $\rho_k < 1$ ensures the existence of a stationary distribution for the (M/M/1) system. The global condition on total service capacity ensures that available processing power exceeds total input intensity.This follows from standard results in queuing theory.

From classical (M/M/1) results:

$$L_k = \frac{\rho_k}{1-\rho_k}, \quad W_k = \frac{1}{\mu_k - \lambda_k} = \frac{L_k}{\lambda_k}, \tag{1}$$

where (W_k) is the expected total waiting time (including service) for key k.

Let p_{loss} denote the probability of event loss (assumed i.i.d.). If an expected event does not arrive within T_{gap}, the algorithm triggers replay attempts, each succeeding with probability p_s.

With at most R allowed attempts, the probability of successful recovery is:

$$P_{fail} = (1 - p_s)^R. \tag{2}$$

The expected additional waiting time due to replay (on successful recovery) is approximately:

$$E[T_{recovery}] \approx T_{gap} + \frac{1-(1-p_s)^R}{p_s} \cdot T^*_{attempt}, \tag{3}$$

where $(T^*_{attempt})$ denotes the average duration between a replay request and response. If events are lost with probability p_{loss}, then the total rate of replay operations for the system (with total input rate $\Lambda = \sum_k \lambda_k$) is:

$$O_{replay} \approx \Lambda \cdot p_{loss} \cdot E[\text{number of attempts per lost event}] \approx \Lambda \cdot p_{loss} \cdot \frac{1}{p_s}. \tag{4}$$

Corollary.

System designers must select p_{loss}, p_s, and T_{gap} such that O_{replay} remains within acceptable limits while avoiding unnecessary replay triggers from false gap detections.

If for a key k an event with sequence number s cannot be recovered after all R replay attempts, then the probability of permanent loss is:

$$P_{fail} = (1-p_s)^R. \tag{5}$$

This failure probability allows engineers to choose the number of replay attempts R that keeps P_{fail} below an acceptable reliability threshold, without excessive replay overhead.

In practice, the number of possible keys $|K|$ can be very large, while available resources are limited: only up to (K_{max}) active internal queues can be maintained at once (due to memory, thread, or descriptor constraints).

The optimization goal is to minimize average waiting time across keys while respecting the resource limit:

$$\min_{\pi} C(\pi) = \sum_k w_k W_k(\pi),\ |\mathrm{A}(\pi,t)| \le K_{max}, \tag{6}$$

where (w_k) are priority weights and $A(\pi, t)$ is the set of active queues under policy (π).

This discrete optimization problem is NP-hard, equivalent to resource allocation or container scheduling.

Practical solutions include heuristic approaches such as LRU (Least Recently Used), TTL-based eviction, or heavy-hitter detection strategies to prioritize high-traffic keys.

In scenarios where a subset of keys exhibits very high event rates (λ_k)), the local stability condition $(\rho_k<1)$ may be violated. Such "hot keys" dominate latency and can monopolize resources, causing overflows in (Q_k).

As total input intensity $\Lambda \to \infty$ with fixed resources (K_{max}, m), average latency $W_k \to \infty$ Hence, scalability requires dynamic resource allocation based on observed load distributions λ_k.

Without gaps, the average waiting time is $W_k^{(0)} = \frac{1}{\mu_k - \lambda_k}$. With gap events, where a fraction p_{loss} of events trigger recovery with mean delay $\mathrm{E}[T_{\mathrm{recovery}}]$, the expected waiting time becomes:

$$W_k = W_k^{(0)} + p_{\mathrm{loss}} \cdot \mathrm{E}[T_{\mathrm{recovery}}]. \tag{7}$$

This linear correction holds when gap events are rare and independent. For correlated or bursty losses, a more sophisticated Markovian or phase-type model must be used.

If losses or delays are correlated over time (e.g., due to network outages or broker overloads), then the independent-loss assumption fails. In this case, a two-state Markov channel (Good/Bad) can be introduced, where transition probabilities define periods of reliability and failure. Replay success probability (p_s) then depends on the current channel state. This extension can be analyzed using balance equations or matrix-analytic methods for phase-type distributions.

The developed mathematical framework provides a comprehensive basis for ensuring the correctness, stability, and efficiency of event processing in microservice systems that use per-key ordering. The algorithm, implemented at the application level through X-Sequence-ID tagging and the gap-replay mechanism, guarantees strict per-key FIFO execution, while maintaining system liveness under the condition of reliable event persistence. Stability and finite latency are achieved when each key's load factor $\rho_k < 1$ and the overall service capacity exceeds incoming traffic. The replay overhead grows linearly with the probability of message loss and decreases with higher replay success rates, allowing quantitative tuning of system parameters. Managing per-key queues is formalized as an optimization problem constrained by available computational resources and can be effectively addressed through adaptive heuristic strategies that balance throughput, latency, and fault tolerance.

Hence, the developed mathematical framework serves as a foundation for a unified approach to per-order parallel event processing that not only models the system behavior theoretically but also defines practical implementation principles. This transition provides a logical bridge from analytical modeling to the description of the method itself—its architecture, mechanisms, and realization within microservice-based event delivery systems.

The proposed per-order ordered parallelism method in microservice-based event delivery systems represents both a scientific and a practical contribution, as it combines a formal model of event ordering with an applied technology capable of maintaining a high level of concurrency without violating execution order.

The scientific result lies in the further development of the theory of asynchronous message processing in distributed systems through the introduction of an *application-level ordering layer* built on top of a message broker such as RabbitMQ. A new approach is proposed, in which the strict execution order of operations is guaranteed not by the transport layer itself but by the application logic of the system, which marks each event with a sequence identifier (X-Sequence-ID) and processes them according to the local state of each order. This makes it possible, for the first time, to combine the properties of total ordering within a single JobKey with multithreaded execution, where different orders are processed in parallel but each retains its own deterministic sequence.

The scientific novelty of the method consists in the formulation of the concept of dynamic per-key queue creation – internal queues that are instantiated at the moment the first event for a specific order is received and are automatically terminated upon order completion. This approach analytically guarantees local ordering without requiring a centralized coordinator. An additional element of novelty is the introduction of a gap-replay mechanism, which uses a persistent JobStates storage to restore missing steps based on historical processing data. The relationship between event sequence numbers and system states is formally defined, allowing one to prove order correctness even in the presence of lost or redelivered messages. As a result, the method ensures the property of exactly-once order correctness at the application protocol level without any modifications to the message broker configuration.

The practical result lies in the creation of a technology that guarantees a strict execution order for each individual order while maintaining high system throughput. Implementation of the method enables reduced latency in event processing, more efficient resource utilization, and automatic recovery after failures. The integration of dynamic per-key queues and the gap-replay mechanism, together with event marking via X-Sequence-ID, makes it possible to preserve state consistency in real time and to avoid errors caused by lost or delayed information. In practice, this means that users experience stable service behavior without duplicate, missing, or out-of-order operations, while the system itself can scale dynamically without loss of reliability.

The proposed per-order ordered parallelism method represents a new approach to ensuring event sequence integrity in distributed microservice systems. It combines application-level ordering, dynamic internal queues, and an automatic gap recovery mechanism, which together improve performance, resilience, and the functional reliability of modern event delivery architectures.

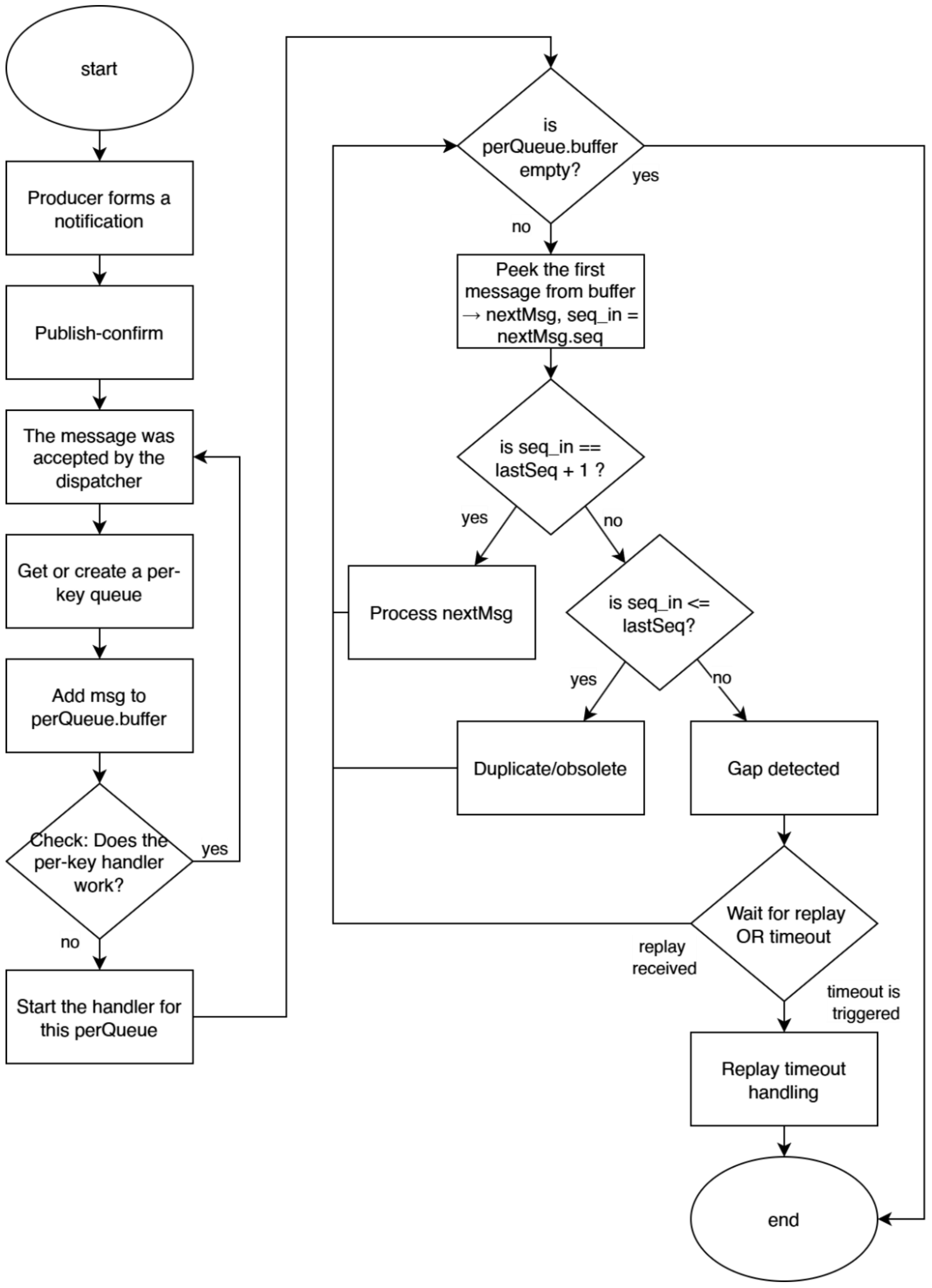
start
Producer forms a notification
Publish-confirm
The message was accepted by the dispatcher
Get or create a per-key queue
Add msg to perQueue.buffer
Check: Does the per-key handler work?
yes
no
Start the handler for this perQueue
is perQueue.buffer empty?
yes
no
Peek the first message from buffer → nextMsg, seq_in = nextMsg.seq
is seq_in == lastSeq + 1 ?
yes
no
Process nextMsg
is seq_in <= lastSeq?
yes
no
Duplicate/obsolete
Gap detected
Wait for replay OR timeout
replay received
timeout is triggered
Replay timeout handling
end

Fig. 1. Per-order ordered parallelism method in microservice event delivery systems

Conclusions and recommendations

As a result of the conducted research, a per-order ordered parallelism method for microservice-based event delivery systems has been proposed. The method ensures the preservation of a strict execution order within each logical order while maintaining high overall system throughput. The introduction of an application-level ordering mechanism built on top of the RabbitMQ message broker, incorporating X-Sequence-ID event tagging, dynamic per-key queue creation, and the gap-replay mechanism for recovering missing states, made it possible to achieve a combination of high concurrency and deterministic execution. Experimental results confirmed the efficiency of the proposed method in building fault-tolerant distributed services where the order of business process steps is critical.

Based on this method, an analytical model of the impact of parallelism on performance and latency in ordered event delivery systems with at-least-once semantics has been developed. The scientific novelty of this model lies in accounting for idempotency, message redelivery, buffering of missing states, and the presence of a backup HTTP transmission channel. Unlike known message queue models, the proposed analytical model enables formal evaluation of overhead ε, determination of the optimal parallelism level, and prediction of system behavior during broker failures or service updates. This contributes to improving the stability and scalability of the AutoGivex microservice architecture and provides a foundation for analytically guided configuration tuning.

The practical implementation of the method enhances system stability, reduces event processing latency, ensures idempotency, and simplifies recovery mechanisms after failures. This makes the approach applicable to financial, logistics, educational, and enterprise platforms where accuracy and timeliness of event processing are essential.

Future research should focus on expanding the analytical model to support multi-level event processing scenarios, refining mathematical relationships between parallelism parameters, latency, and throughput, and developing formal verification methods for order correctness under large-scale distributed conditions. Further work should also aim to optimize the gap-replay process to minimize recovery time, integrate blockchain technologies to improve the trustworthiness of event logs, and implement adaptive real-time load-balancing strategies among per-key queues to maximize performance and reliability.

References

1. RabbitMQ, "Consumer Acknowledgements and Publisher Confirms," RabbitMQ Documentation. [Online]. Available: https://www.rabbitmq.com/docs/confirms. [Accessed: 19-Oct-2025]. rabbitmq.com

2. RabbitMQ, "Reliable Publishing with Publisher Confirms," RabbitMQ Tutorials, Tutorial 7. [Online]. Available: https://www.rabbitmq.com/tutorials/tutorial-seven-java.html. [Accessed: 19-Oct-2025]. rabbitmq.com

3. RabbitMQ, "Publishers – Strategies for Using Publisher Confirms," RabbitMQ Documentation. [Online]. Available: https://www.rabbitmq.com/docs/publishers.html. [Accessed: 19-Oct-2025]. rabbitmq.com

4. C. Richardson, "Transactional Outbox Pattern," Microservices.io. [Online]. Available: https://microservices.io/patterns/data/transactional-outbox.html. [Accessed: 19-Oct-2025]. microservices.io

5. G. Morling, "Revisiting the Outbox Pattern," Decodable Blog, Oct. 31, 2024. [Online]. Available: https://www.decodable.co/blog/revisiting-the-outbox-pattern. [Accessed: 19-Oct-2025]. decodable.co

6. J. Kreps, "Exactly-Once Semantics Are Possible: Here's How Apache Kafka Does It," Confluent Blog, Jun. 30, 2017. [Online]. Available: https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/. [Accessed: 19-Oct-2025]

7. Confluent, "Message Delivery Guarantees for Apache Kafka," Confluent Documentation. [Online]. Available: https://docs.confluent.io/kafka/design/delivery-semantics.html. [Accessed: 19-Oct-2025]. docs.confluent.io

8. Apache Kafka, "Kafka Documentation," Apache Kafka Project. [Online]. Available: https://kafka.apache.org/documentation/. [Accessed: 19-Oct-2025]. Apache Kafka

9. CloudAMQP Blog, "RabbitMQ: Exchanges, routing keys and bindings," Sep. 24, 2019. [Online]. Available: https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html. [Accessed: 19-Oct-2025]. CloudAMQP

10. Baeldung, "Consumer Acknowledgments and Publisher Confirms with RabbitMQ," Jul. 8, 2024. [Online]. Available: https://www.baeldung.com/rabbitmq-consumer-acknowledgments-publisher-confirmations. [Accessed: 19-Oct-2025]. Baeldung on Kotlin

11. SAP Community, "Message Replay (SAP Event Mesh)," SAP Community / Event Mesh. [Online]. Available: https://community.sap.com/t5/technology-blog-posts-by-members/reversing-the-time-with-advanced-event-mesh-unleash-the-power-of-message/ba-p/13548246 (overview) and https://help.pubsub.em.services.cloud.sap/Features/Replay/Message-Replay-Overview.htm (feature). [Accessed: 19-Oct-2025]. community.sap.com+1

12. R. Laigner, A. C. Almeida, W. K. G. Assunção, Y. Zhou, "An Empirical Study on Challenges of Event Management in Microservice Architectures," arXiv:2408.00440, Aug. 2024. [Online]. Available: https://arxiv.org/abs/2408.00440. [Accessed: 19-Oct-2025]. arXiv

13. S. Kumar, A. Jadon, S. Sharma, "Global Message Ordering using Distributed Kafka Clusters," arXiv:2309.04918, Sep. 2023. [Online]. Available: https://arxiv.org/abs/2309.04918. [Accessed: 19-Oct-2025]. arXiv

14. "SoK: Consensus for Fair Message Ordering," arXiv:2411.09981, Nov. 2024 (updated Jun. 2025). [Online]. Available: https://arxiv.org/abs/2411.09981. [Accessed: 19-Oct-2025]. arXiv

15. Microservices.io, "Pattern: Transaction log tailing," Microservices Patterns. [Online]. Available: https://microservices.io/patterns/data/transaction-log-tailing.html. [Accessed: 19-Oct-2025]. microservices.io

16. Microservices.io, "Patterns for Microservices Architecture," Chris Richardson. [Online]. Available: https://microservices.io/patterns/. [Accessed: 19-Oct-2025]. microservices.io

17. AWS Builders' Library, "Making retries safe with idempotent APIs," Amazon Web Services. [Online]. Available: https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/. [Accessed: 19-Oct-2025]. Amazon Web Services, Inc.

18. "Service-outbox: Transactional outbox pattern for microservices," GitHub Repository. [Online]. Available: https://github.com/yornaath/service-outbox. [Accessed: 19-Oct-2025]. GitHub

19. "A Benchmark for Data Management in Microservices," arXiv:2403.12605, Mar. 2024. [Online]. Available: https://arxiv.org/abs/2403.12605. [Accessed: 19-Oct-2025]. arXiv

20. SoftwareMill, "Microservices 101: Transactional Outbox and Inbox," SoftwareMill blog, Jun. 3, 2022. [Online]. Available: https://softwaremill.com/microservices-101/. [Accessed: 19-Oct-2025]. SoftwareMill

21. Queues | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/queues?utm_source=chatgpt.com

22. Narkhede N. Exactly-once Semantics is Possible: Here's How Apache Kafka Does it. Confluent. URL: https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/?utm_source=chatgpt.com

А. В. Колодюк, О. Б. Волощук

МЕТОД УПОРЯДКОВАНОГО ПАРАЛЕЛІЗМУ PER-ЗАМОВЛЕННЯ У МІКРОСЕРВІСНИХ СИСТЕМАХ ДОСТАВКИ ПОДІЙ

У статті представлено метод упорядкованого паралелізму per-order для систем доставки подій у мікросервісних архітектурах, спрямований на забезпечення строгої послідовності виконання операцій у межах кожного логічного замовлення за умов високої інтенсивності подій та відмовостійкості. Запропонований підхід ґрунтується на впровадженні прикладного рівня впорядкування поверх брокера повідомлень RabbitMQ без потреби модифікації його внутрішніх механізмів. Метод включає маркування подій ідентифікаторами послідовності X-Sequence-ID, динамічне створення внутрішніх черг per-key та механізм відновлення пропущених станів gap-replay, що забезпечує гарантоване відтворення втраченої події і точне відновлення локального стану обробки.

Наукова новизна полягає у формалізації детермінованого механізму обробки подій із підтримкою at-least-once семантики, ідіомою ідемпотентності та контролем редоставки. У рамках роботи побудовано аналітичну модель, яка описує вплив рівня паралелізму на пропускну здатність, латентність та стабільність системи з урахуванням імовірності втрат повідомлень, навантаження на черги, обмеження ресурсів та поведінки системи під час часткових відмов брокера. У моделі враховано відмовостійкий HTTP-канал резервної доставки, що дає можливість формально оцінити накладні витрати та передбачити динаміку системи під час пікових навантажень.

Експериментальна перевірка методу у промисловій мікросервісній архітектурі AutoGivex показала зниження затримки, підвищення стабільності виконання, зменшення випадків розупорядкування та покращення масштабованості без втручання у транспортний шар. Результати демонструють, що запропонований підхід придатний для побудови високонавантажених розподілених систем, де критичною є узгодженість станів і точність виконання бізнес-процесів. Отримані аналітичні висновки можуть бути використані як підґрунтя для адаптивної оптимізації паралелізму, конфігурації черг та автоматизованого керування продуктивністю.

Ключові слова: впорядкований паралелізм; мікросервіси; доставка подій; RabbitMQ; X-Sequence-ID; ідемпотентність; gap-replay; модель паралельності; відмовостійкість; аналітична модель; AutoGivex; інформаційні технології.