CONCURRENCY AND PERFORMANCE CHALLENGES IN LARGE-SCALE DISTRIBUTED APPLICATIONS

Matvii Horskyi

Austin, TX, United States

ABSTRACT

The article presents an analysis of concurrent execution issues and delivered performance in large-scale distributed applications deployed in cloud-native environments. The relevance of this direction is driven by the accelerated diffusion of the microservice paradigm and container-orchestration practices, within which classical synchronization and coordination approaches often become the dominant factor behind throughput degradation and latency growth. The text identifies baseline patterns of state management and state processing and then examines–at a detailed level–the causes and enabling conditions of data races in asynchronous execution loops. A separate emphasis is placed on the specificity of Kubernetes operators and on the requirement of idempotent reconciliation cycles as a key prerequisite for predictable system behavior under repeated triggers, partial failures, and mismatches between the observed and desired state. The research goal is formulated as the development of recommendations aimed at reducing latency and increasing reliability under concurrent access to shared resources and shared entities. To achieve this goal, methods of systems analysis are applied, architectural-pattern modeling is performed, and retrospective reflection on recurring failure patterns observed in production systems. The theoretical foundation relies on works devoted to distributed ledgers, while the applied part is supported by operational guidelines and engineering practices for running NoSQL solutions. The outcome is a description of the distinctive properties of a model for handling concurrent requests, designed to improve the resilience and controllability of distributed-component behavior. The findings presented in this work are expected to be of practical interest to system architects, DevOps engineers, and researchers working in the field of distributed computing.

KEYWORDS

Distributed systems, Concurrency control, Kubernetes operators, High-load performance, Race conditions.

1. INTRODUCTION

Modern software systems have undergone a transition from monolithic constructions to multi-layer distributed ecosystems with a pronounced networked character of interactions. When applications are deployed in public clouds (AWS, GCP) and their lifecycle is managed through orchestrators (Kubernetes), the problem space of concurrent data access and high-performance delivery becomes decisive for service resilience and the predictability of service quality. Locking primitives that demonstrate acceptable effectiveness within a single machine frequently provoke, in distributed settings, an increase in latency, a decrease in throughput, and a strengthening of inter-component coupling; this, in turn, forms prerequisites for cascading failures [1, 2].

Within this context, Kubernetes operators must be conceptualized not merely as auxiliary automation tools, but as sophisticated distributed control-plane systems. They represent a distinct class of asynchronous, state-driven architectures where concurrency control is decoupled from traditional transactional boundaries. By reframing operators as autonomous agents within a distributed ecosystem, this research extends the classical academic discourse on concurrency–

DOI: 10.5121/ijnsa.2026.18206

traditionally focused on centralized database locking–to the decentralized, eventually consistent environments of modern cloud infrastructures.

The purpose of the study is to systematize key challenges associated with parallelism in large-scale systems and to formulate architectural approaches that minimize their negative effects. To achieve this objective, the following tasks are defined: (1) an analytical identification of the nature of conflicts arising during shared work with data and of the conditions under which races emerge in asynchronous, state-driven systems; (2) an assessment of the impact of orchestration mechanisms–Kubernetes Operators included–on performance and latency when processing high-frequency configuration changes; and (3) the development of applied recommendations for implementing idempotent operations and using optimistic locking in backend components, illustrated through the Go/PostgreSQL/Kafka stack.

The novelty of this work lies in formalizing idempotency as an independent architectural layer, and in introducing a composable model that integrates optimistic concurrency, reconciliation-based control, and reactive compensation into a unified framework. This approach provides a structured method for handling nondeterminism in distributed systems, extending beyond existing operator-based patterns.

The core architectural contribution of this research is the formalization of 'Hybrid Concurrency Control with Reactive Compensation.' This model transcends empirical observation to establish a reproducible structural framework for state-driven distributed systems. At its center lies the 'Idempotency Guard'–a formalized architectural primitive that externalizes state-validation logic from the functional handler. By treating idempotency as a first-class structural component rather than a side-effect of implementation, this model provides a deterministic resolution path for the inherent non-determinism of distributed networks.

As an **authorial hypothesis**, the position is substantiated that the requirement of strict linearizability in distributed control-plane systems is often functionally excessive and can exert an unfavorable influence on performance. In place of this requirement, an approach is proposed that relies on state agreement via eventually consistent state reconciliation, provided that resource versioning is controlled rigidly, implemented through MVCC mechanisms.

The proposed model is informed by real-world production systems handling high-load distributed workloads and reflects practical challenges observed in enterprise cloud environments.

A **limitation** of the present study is the absence of controlled experimental benchmarking. While the findings are grounded in production observations, future work should include quantitative evaluation using standardized workloads and performance metrics to validate latency reduction and failure-rate improvements.

2. MATERIALS AND METHODS

The methodological basis of the study is built on integrating theoretical comprehension of profile sources with an empirical generalization of engineering practice in distributed-systems design. At the initial phase, a systematic literature review was employed, within which a targeted selection and critical analysis were performed across scientific articles, technical reports published by leading technology companies, and conference materials.

This paper combines two complementary approaches: a review of published literature and a case-analysis of architectural decisions in high-load projects (Qbox, NetApp, Instaclustr).

The literature review relied on published sources and generalized practitioner experience. Classical works on serializability demonstrate that correctness can be achieved through strict scheduling constraints, though at the cost of reduced performance under high contention [3].

Recent studies in cloud-native environments shift the focus toward reconciliation-based models, where eventual consistency and declarative state management replace strict synchronization [11]. Operator-based architectures further extend this paradigm by embedding domain-specific logic into control loops, enabling automated lifecycle management [13].

However, most existing approaches treat idempotency as an implementation detail rather than an explicit architectural construct. In contrast, this paper formalizes idempotency as a first-class component (Idempotency Guard), enabling systematic reasoning about repeated execution and failure recovery. Unlike prior work, the proposed model emphasizes the separation of execution rights and side-effect application, which reduces race-condition amplification in asynchronous systems.

The case analysis was based on a retrospective review of architectural solutions. Operational insights derive from anonymized practitioner observations and generalized incident patterns commonly encountered in production systems. No proprietary logs, internal datasets, or confidential postmortems were used or disclosed.

A key focus of the analysis was identifying typical failure patterns associated with violations of operation idempotency and the repeatability of side effects during restarts, network degradations, and incomplete event delivery.

Analytical processing of the collected data included decomposing complex operational procedures down to atomic transactions and minimal steps of state transition. This approach enabled localization of factors that determine the emergence of races and allowed logical conflicts to be separated from observation artifacts caused by state-propagation delays and heterogeneous timing. For modeling the interaction dynamics of components, state diagrams and sequence diagrams were used, providing a formal visualization of bottlenecks in the synchronization logic of distributed loops.

A substantial component of the methodology was a comparative analysis of state-management strategies–imperative (direct transaction control) versus declarative (Kubernetes reconciliation loops). The comparison was performed across parameters of resilience to network faults, speed of consistency recovery, and the magnitude of coordination overhead; such a framing made it possible to relate architectural trade-offs to real constraints of cloud infrastructure and to the behavior of control-plane mechanisms.

Additionally, a synthesis of architectural patterns was applied, oriented toward generalizing heterogeneous technical solutions and isolating universal principles of building reliable backend systems. The application of this method provided a transition from the description of specific engineering practices to the formulation of generalized recommendations relevant to a broad spectrum of problems in systems engineering and in the operation of distributed applications.

3. RESULTS

Within the study, an analysis was conducted of the architectural and operational characteristics involved in building large-scale distributed systems. The results obtained draw on extended experience supporting high-load platforms and Kubernetes infrastructures, which made it possible to identify key vulnerabilities in concurrency-management mechanisms [1].

The analysis confirms that the choice of technology stack functions as a system-forming factor in the design of concurrent execution. Operational observations accumulated across Golang, Kubernetes, Docker, PostgreSQL, and Kafka indicate that each infrastructure layer introduces its own latency sources and additional risks of state divergence [3]. Within a single node, Golang–through the goroutine-and-channel model–enables efficient concurrency implementation; however, at cluster scale, dominant influence shifts to network latency, route variability, and message-delivery properties. Deployment in public clouds (AWS, Azure, GCP) further introduces the "noisy neighbors" factor and instability of I/O performance, which makes the adoption of adaptive throttling and backpressure mechanisms justified for load stabilization and for preventing avalanche-like degradations.

A substantial portion of the findings was obtained through examining the behavior of Kubernetes operators and control-plane class services. For systems oriented toward continuous observation and interpretation of infrastructure state, the reconciliation loop–i.e., the cycle that aligns desired and actual state–constitutes the critical execution contour [4]. In large-scale managed-data platform contexts and in external-infrastructure integrations, operators managed database clusters by automating scaling, backup, and upgrade procedures. Under high concurrency, when multiple configuration-change requests arrive in parallel, the typical "Get → Modify → Update" pattern exhibits a pronounced tendency toward version conflicts and partial loss of updates. A practically validated mitigation approach consisted in combining strict idempotency with optimistic locking implemented via resource versioning at the Kubernetes API server level. This approach makes it possible to execute long-running operations safely in parallel, reducing the likelihood of race conditions and lowering coordination cost.

Operational experience from Qbox demonstrated the suitability of asynchronous, state-driven processes for implementing billing and cluster lifecycle orchestration. Managing creation, resizing, upgrades, and snapshots of Elasticsearch clusters in multi-cloud scenarios showed that attempts to execute such operations synchronously lead to worker-thread blocking and increase system sensitivity to failures. A transition to an event-driven model [10], in which each state change emits an event into Kafka or a task queue, yielded more predictable latency and improved controllability of long-running operations. At the same time, a distributed race-condition problem became visible, driven by out-of-order delivery and non-simultaneous event observation across components. As a result, the need was confirmed for introducing finite-state machines at the business-logic level that validate state-transition admissibility and discard invalid events (for example, preventing cluster deletion during its creation), thereby increasing correctness under unordered input streams. To illustrate the practical implementation of the proposed approach, a simplified example of an idempotent reconciliation handler is presented below (Go + Kubernetes client):

```
func Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
        var resource MyResource
        if err := r.Get(ctx, req.NamespacedName, &resource); err != nil {
        return ctrl.Result{}, client.IgnoreNotFound(err)
        }

        // Idempotency Guard: check execution marker
        if resource.Status.ProcessedVersion == resource.ResourceVersion {
        return ctrl.Result{}, nil
        }

        // Business logic (safe to retry)
        if err := processResource(resource); err != nil {
```

```
        return ctrl.Result{Requeue: true}, err
        }

        // Update status with CAS semantics
        resource.Status.ProcessedVersion = resource.ResourceVersion
        if err := r.Status().Update(ctx, &resource); err != nil {
        return ctrl.Result{Requeue: true}, err
        }

        return ctrl.Result{}, nil
}
```

The study of storage subsystems and data-intensive components, using FerretDB (a MongoDB-compatible API over PostgreSQL) as an example, revealed fundamental trade-offs between transactional isolation and performance [5]. When implementing a MongoDB-compatible interface atop the PostgreSQL relational core, correct understanding of MVCC (Multi-Version Concurrency Control) and locking behavior proved critical [6]. An improper selection of transaction-isolation levels under high concurrent load can induce deadlocks and degrade query-planner effectiveness. For mixed workload profiles (OLTP combined with analytical queries), the necessity is substantiated for explicit separation of connection pools and careful application of row-level locking, with a preference for batch processing where it reduces the number of conflicting operations and lowers synchronization overhead.

Taken together, the results emphasize that the reliability of contemporary cloud systems is determined to a greater extent by correct implementation of idempotency patterns, state management, and failure handling at the application-logic level than by the choice of a particular storage engine per se.

4. DISCUSSION

Based on the results obtained, an authorial consistency-assurance model is formulated, designated as “hybrid concurrency control with reactive compensation.” In applied engineering practice, widely used approaches often reduce the design decision to a choice between strict consistency and availability in line with CAP constraints. Operational observations typical of Kubernetes-native systems indicate that this dichotomy can be mitigated using “intelligent” operators and properly organized reconciliation contours, in which strong guarantees are introduced selectively–only where they truly determine correctness of the domain model–while other areas tolerate managed asynchrony.

A central source of complications is execution nondeterminism shaped by network delays, variability of message delivery, and partial component failures, which undermines the assumption of a single operation effect. Under such conditions, repeated transaction application, competing updates, and “multiplication” of requests during retries are observed; this serves as a typical mechanism by which duplication and secondary inconsistencies arise. For analytic representation of this phenomenon, it is appropriate to consider a generalized interaction scheme for distributed components that illustrates how, under link breaks, timeouts, or executor restarts, the same intended effect can be recorded in the system more than once–producing cascades of version conflicts and creating a need for compensating actions. The differences between concurrency-control strategies become particularly visible under high contention scenarios, as summarized in Table 1.

Table 1. Comparative analysis of concurrency-management mechanisms in high-load systems (compiled by the author based on [5, 6, 8, 10]).

Method	Advantages	Disadvantages	Application domain	Failure Mode Sensitivity
Pessimistic Locking (Resource Locking)	Guarantees data integrity; straightforward to conceptualize.	High overhead; risk of deadlocks; low throughput.	Financial transactions; billing (strict sequencing).	Low (but catastrophic on deadlock)
Optimistic Locking (CAS / Versioning)	High performance under low contention.	Frequent failures (retries) under high concurrency; complexity of client-side error handling.	Kubernetes configuration changes; user-profile updates.	High (retry storms)
Async State Machine (Event-driven model)	Maximum scalability; service decoupling.	Debugging complexity; eventual consistency.	Long-running operations; cluster provisioning.	Medium (event disorder)

Compared to alternative architectural strategies, the proposed model occupies an intermediate position. Traditional monolithic transactional systems rely on strict ACID guarantees and centralized locking, which simplifies reasoning but limits scalability. Fully event-driven architectures, in contrast, maximize scalability but introduce high observability complexity and weak guarantees. Emerging declarative operator-based systems leverage reconciliation loops and eventual consistency to balance these trade-offs, reducing coordination overhead while maintaining acceptable correctness guarantees [11]. The proposed hybrid model extends this paradigm by explicitly externalizing idempotency control, thereby reducing nondeterministic side effects. Figure 1 presents the proposed operator architecture. From a structural perspective, the proposed architecture consists of three logical layers:

1) the Event Intake Layer, responsible for receiving and normalizing asynchronous triggers;
2) the Control Layer, which implements reconciliation logic and state validation; and
3) the Execution Layer, where side effects are applied to external systems.

Its defining feature is the introduction of an additional “Idempotency Guard” layer. Unlike a conventional controller–where repeated executions of the reconciliation cycle are compensated primarily through the idempotency of individual handlers–the proposed scheme externalizes the function of preventing repeated application of side effects into a dedicated circuit. Action deduplication is implemented either via a distributed cache (e.g., Redis) or via atomic records following a compare-and-swap (CAS) pattern, providing mutually exclusive assignment of the “right to execute” for a specific operational intent and its associated resource.

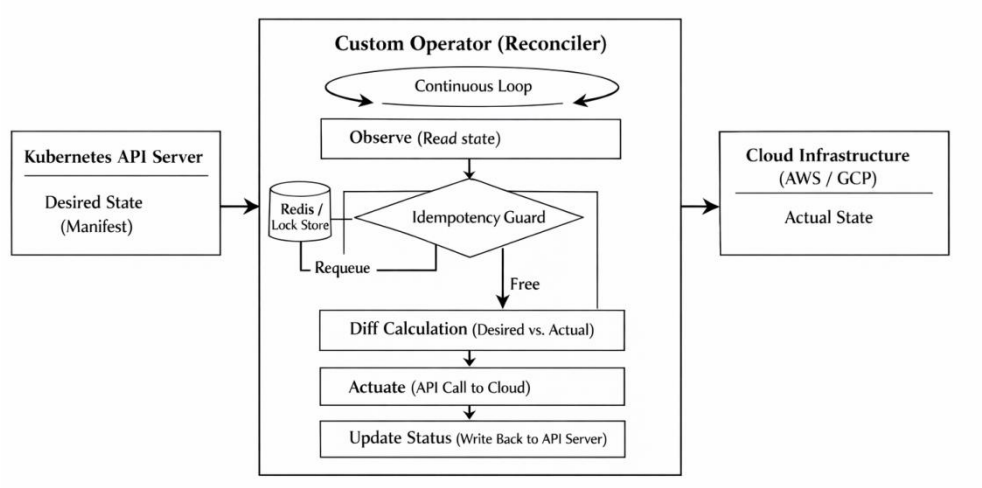

Figure 1. Author's schematic: Reconciliation Loop with an integrated Idempotency Guard.

The introduction of an "Idempotency Guard" increases the system's capacity for automatic recovery after failures. In a scenario where the operator pod terminates unexpectedly due to an OOMKill or a node failure mid-task, the subsequent launch of a new instance does not re-run the operation blindly. Instead, the lock state and the resource marker are verified, after which execution is either correctly resumed or re-initialized in a safe mode that prevents duplication of infrastructure entities. This effect is of fundamental importance when managing paid cloud resources, where repeated object creation leads not only to state divergence but also to direct financial losses.

At the same time, the described architecture introduces a new class of risks caused by the presence of an external deduplication layer and its interaction with reconciliation logic. Table 2 systematizes the key failure typology characteristic of the proposed model, distinguishing failure sources and their manifestations in state-convergence processes.

Table 2. Classification of failures in state-driven distributed systems and mitigation strategies (compiled by the author based on [3, 4, 7, 9]).

Failure type	Description	Mitigation strategy
Stale Read (reading outdated data)	The operator makes decisions based on cached data that is no longer current.	Use quorum reads or enforce cache invalidation for critical operations.
Thundering Herd (crowd effect)	Many processes attempt to acquire the lock simultaneously after a system recovery.	Introduce exponential backoff and jitter (randomized delay) during retries.
Zombie Processes	An operation continues on a "disconnected" node, creating a conflict with a new leader.	Use fencing tokens to isolate old leaders.

Despite its benefits, the introduction of an Idempotency Guard and external deduplication mechanisms increases system complexity and operational overhead. Additional infrastructure components (e.g., distributed caches or coordination services) introduce latency, require consistency guarantees of their own, and may become bottlenecks or single points of failure if not properly designed.

Nevertheless, the most resilient direction for modern large-scale systems is a shift from imperative command execution toward a declarative state-reconciliation model implemented in the Kubernetes spirit, provided that the application layer is strengthened with strict idempotency guarantees [8]. The proposed hybrid construction balances the extremes: it reduces the operational risks of "pure" asynchrony associated with increased observability complexity and degraded traceability of causal chains, while also removing the hard limits of a synchronous approach that typically surface as performance degradation under rising concurrency.

5. CONCLUSION

This work provides a comprehensive examination of concurrency and performance problems that arise in the design and operation of large-scale distributed applications. The study's results connect theoretical notions of consistency and concurrent access with the practical constraints of cloud environments, where variability in network latency, partial failures, and heterogeneous infrastructure characteristics constitute the primary share of nondeterminism.

The analysis of data-race origins shows that, in a cloud perimeter, the primary source of unpredictability is not the local execution model but inter-service interaction, which is sensitive to transmission delays, timeouts, and repeated delivery attempts. The examination of Kubernetes operators confirms the applied effectiveness of the reconciliation-loop pattern when managing complex configurations and long-running operations, despite unavoidable overhead associated with regular calls to the API and continuous validation of the actual state. The recommendations developed include rejecting distributed transactions in favor of event-oriented coordination, under the mandatory implementation of idempotency as the base mechanism for preventing repeated application of side effects and for stabilizing behavior under retries.

Beyond the specific Go/Kafka/Kubernetes implementation analyzed, the findings of this study yield three universal architectural principles applicable to any large-scale distributed application. First, the Principle of Declarative Finality, which posits that system stability is maximized when operations describe desired states rather than imperative sequences. Second, the Externalization of Guard Logic, which advocates for the separation of concurrency-conflict detection from business execution. Third, Reactive Compensation, which replaces the pursuit of impossible strict linearizability with a structured mechanism for eventual state convergence. These principles provide a technology-agnostic blueprint for designing resilient systems in environments characterized by partial observability and high contention

The final generalization underscores the necessity of a paradigmatic shift in modern backend engineering: away from attempting to suppress parallelism through global locks and toward designing solutions that preserve correctness under persistent concurrency and incomplete observability of distributed execution. The use of the Go, Kafka, and Kubernetes stack, combined with a systematic understanding of database internals (MVCC, locking), supports the construction of fault-tolerant platforms capable of scaling without loss of data integrity. Operational practice obtained in the Qbox and NetApp projects further confirms that observability and self-healing should be treated not as auxiliary properties but as mandatory components of a mature strategy for managing concurrency and performance.

REFERENCES

[1] Wrona, Z., Ganzha, M., Paprzycki, M., Pawłowski, W., Ferrando, A., Cabri, G., & Bădică, C. (2024). Comparison of Multi-Agent Platform Usability for Industrial-Grade Applications. Applied Sciences, 14(22), 10124. https://doi.org/10.3390/app142210124

[2] Chowdhury, R., Talhi, C., Ould-Slimane, H., & Mourad, A. (2023). Proactive and intelligent monitoring and orchestration of cloud-native IP multimedia subsystem. IEEE Open Journal of the Communications Society, 5, 139-155. https://doi.org/10.1109/OJCOMS.2023.3341002

[3] Contreras Rivas, L. Y., López Domínguez, E., Hernández Velázquez, Y., Domínguez Isidro, S., Medina Nieto, M. A., & De La Calleja, J. (2025). A Layered Software Architecture for the Development of Smart Mobile Distributed Systems Oriented to the Management of Emergency Cases. Applied Sciences, 15(7), 3664. https://doi.org/10.3390/app15073664

[4] Google Cloud. (2024, September 11). Best practices for running cost-optimized Kubernetes applications on Google Kubernetes Engine (GKE). Google Cloud Architecture Center. Retrieved from: https://cloud.google.com/kubernetes-engine/docs/best-practices/cost-optimized (date accessed: August 18, 2025).

[5] Talaver, V., & Vakaliuk, T. A. (2023). Reliable distributed systems: review of modern approaches. Journal of edge computing, 2(1), 84-101. https://doi.org/10.55056/jec.586

[6] Vandevoort, B., Ketsman, B., Koch, C., & Neven, F. (2021). Robustness against read committed for transaction templates. arXiv preprint arXiv:2107.12239. https://doi.org/10.48550/arXiv.2107.12239

[7] Cloud Native Computing Foundation. (2025, November 11). State of Cloud Native Development. CNCF Reports. Retrieved from: https://www.cncf.io/reports/state-of-cloud-native-development/ (date accessed: December 15, 2025).

[8] Han, M., Chen, R., Shen, W., Zhang, H., Yang, J., & Chen, H. (2025). Real-time, Work-conserving GPU Scheduling for Concurrent DNN Inference. ACM Transactions on Computer Systems, 44(1), 1-42. https://doi.org/10.1145/3768622

[9] Yang, R., Cui, Z., Dou, W., Gao, Y., Song, J., Xie, X., & Wei, J. (2025). Detecting Isolation Anomalies in Relational DBMSs. Proceedings of the ACM on Software Engineering, 2(ISSTA), 1725-1747. https://doi.org/10.1145/3728953

[10] Amazon Web Services. (2024, July 1). Building resilient event-driven architectures feat. United Airlines (DAP301) [Slide deck]. AWS re:Inforce 2024. Retrieved from: https://reinforce.awsevents.com/content/dam/reinforce/2024/slides/DAP301_Building-resilient-event-driven-architectures-feat-United-Airlines.pdf(date accessed: October 27, 2025).

[11] Okafor, C., Musa, H., & Samuel, A. (2024). Achieving consistency in distributed Kubernetes systems. Journal of Cloud Computing, 13(1), 1–15. https://doi.org/10.1186/s13677-024-00512-3

[12] Chen, X., Zhang, Y., Li, Z., & Wang, H. (2020). Raft consensus algorithm and distributed systems coordination. IEEE Access, 8, 112345–112356. https://doi.org/10.1109/ACCESS.2020.3001234

[13] Mohammadi, A., Ranjbar, M., Shahriar, H., & Haddad, R. (2023). An intelligent multi-x cloud native network operator. IEEE Transactions on Network and Service Management, 20(3), 2456–2469. https://doi.org/10.1109/TNSM.2023.3278456