© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1715957

Designing Fault-Tolerant Event-Driven Systems: A SAGA-Oriented Approach to Consistency in High-Throughput Environments

ILKER KANATLI

Abstract- ***Event-driven architectures have become the foundation for building scalable and responsive enterprise systems. By enabling asynchronous communication and decoupling services, these systems can handle high-throughput workloads and dynamic operational conditions. However, maintaining consistency across distributed components remains a critical challenge, particularly in environments where failures, delays, and retries are inevitable. The SAGA pattern has emerged as a widely adopted solution for managing distributed transactions without relying on strict coordination mechanisms. By decomposing complex operations into smaller steps and introducing compensating actions, SAGA provides a practical framework for achieving eventual consistency. However, in real-world systems, compensations are not always sufficient to guarantee coherent outcomes. This paper introduces a novel perspective on distributed consistency through the concept of a Convergence-Oriented SAGA model. Instead of focusing on execution paths, the proposed approach emphasizes outcome convergence. Systems are designed to continuously reconcile their current state with a desired target state, ensuring that distributed processes move toward consistent outcomes despite failures and uncertainties. The study develops a conceptual and architectural framework for designing fault-tolerant, event-driven systems that prioritize convergence over strict execution control. It demonstrates how this approach enhances resilience, improves consistency, and supports reliable system behavior in high-throughput environments.***

Keywords - Event-Driven Systems, SAGA Pattern, Distributed Systems, Fault Tolerance, Eventual Consistency

I. INTRODUCTION

Event-driven systems have become a foundational paradigm in modern software architecture, particularly in environments that demand scalability, responsiveness, and adaptability. By enabling asynchronous communication and decoupling services, these systems allow organizations to process large volumes of data and respond dynamically to changing workloads.

Event-driven systems have become a natural choice for building scalable and responsive architectures. By decoupling services and allowing them to react to events asynchronously, these systems can handle high throughput and adapt to changing workloads.

Despite these advantages, the distributed nature of event-driven systems introduces significant challenges, particularly in maintaining consistency across independent services. In traditional monolithic systems, consistency can be enforced through centralized transactions and strict control over execution. In distributed environments, however, such mechanisms become impractical due to scalability constraints and system complexity.

One of the most widely adopted approaches to addressing this challenge is the SAGA pattern. SAGA provides a way to manage distributed transactions by breaking them into smaller, independent steps, each of which can be executed and compensated separately.

However, with this flexibility comes a new set of challenges—especially when it comes to maintaining consistency. One of the most widely adopted solutions to this problem is the SAGA pattern. Instead of relying on distributed transactions, SAGA breaks a large operation into smaller steps. Each step is executed independently, and if something goes wrong, compensating actions are triggered to undo previous work. On paper, this provides a practical way to manage consistency in distributed environments. In reality, things are rarely that clean.

While the SAGA pattern provides a practical framework, its real-world implementation reveals limitations that are often overlooked in theoretical models. Distributed systems operate under conditions of uncertainty, where failures, delays, and inconsistencies are not exceptions but expected behaviors. To illustrate this challenge, consider a typical enterprise workflow involving multiple distributed services.

Consider a typical business process such as order fulfillment. A payment is processed, inventory is reserved, and shipment is initiated. Each of these steps may succeed or fail independently, and failures can happen at any point. When they do, compensating actions attempt to roll the system back to a consistent state. But here is the problem: not everything can be undone.

This observation highlights a critical limitation of traditional SAGA implementations. While compensating actions aim to reverse previous steps, they cannot always fully restore the system to its original state. Certain actions, once executed, introduce irreversible effects or external dependencies that cannot be undone through simple compensation.

As a result, distributed systems may exhibit behaviors where individual components operate correctly, yet the overall system state becomes inconsistent or misaligned. This introduces a more subtle and complex problem—one that extends beyond failure handling into the domain of system convergence.

This paper argues that addressing this challenge requires a shift in perspective. Rather than focusing solely on execution and compensation, distributed systems must be designed to ensure that they converge toward a consistent and intended outcome over time.

II. EVOLUTION OF DISTRIBUTED AND EVENT-DRIVEN SYSTEMS

The evolution of distributed systems reflects a continuous effort to balance scalability, availability, and consistency. Early system architectures relied on tightly coupled components and centralized control, where transactional integrity was maintained through ACID-compliant database systems. These systems ensured strong consistency but were limited in their ability to scale horizontally and adapt to dynamic workloads.

With the increasing demand for scalability, distributed architectures emerged, introducing new models such as Service-Oriented Architecture (SOA) and later microservices. These paradigms emphasized modularity, loose coupling, and independent deployment of services. While these changes improved system flexibility and scalability, they also introduced new challenges in maintaining coordination and consistency across distributed components.

Event-driven architectures further extended this evolution by enabling asynchronous communication between services. Instead of relying on synchronous request-response interactions, services communicate through events, allowing systems to react dynamically to changes. This approach enhances scalability and resilience, particularly in high-throughput environments where systems must handle large volumes of concurrent operations.

However, this shift comes at a cost. As systems become more distributed and interactions more asynchronous, maintaining a consistent global state becomes increasingly difficult. Unlike centralized systems, where transactions can enforce atomicity, distributed systems must operate under conditions of partial failure, network latency, and eventual consistency.

The challenge is further amplified in high-throughput environments, where the volume and velocity of events introduce additional complexity. Events may arrive out of order, be duplicated, or be delayed. Services may process events at different speeds, leading to temporal inconsistencies across the system. These factors make it difficult to ensure that all components reflect a coherent state at any given time.

A payment may already be captured. A shipment may already be in transit. A retry may trigger the same step multiple times. Delays between services can

cause actions to arrive out of order. Over time, these factors create a system where different parts drift away from each other, even though each individual service behaves correctly.

This phenomenon highlights a fundamental characteristic of distributed systems: correctness at the component level does not guarantee correctness at the system level. Each service may execute its responsibilities accurately, yet the overall system may exhibit inconsistencies due to asynchronous interactions and partial failures. The result is a subtle but critical issue. The system does not just fail—it diverges.

This concept of divergence represents a shift from traditional notions of failure. In many cases, distributed systems do not experience catastrophic breakdowns. Instead, they gradually move away from a consistent state, accumulating inconsistencies over time. These inconsistencies may remain undetected until they manifest as operational issues, making them particularly difficult to diagnose and resolve.

Traditional approaches to managing consistency, including distributed transactions and SAGA-based coordination, attempt to address these challenges by controlling execution flow and compensating for failures. However, as system complexity increases, these approaches face limitations in ensuring that all execution paths lead to consistent outcomes. The evolution of distributed and event-driven systems thus sets the stage for a deeper examination of consistency models and coordination mechanisms. Understanding the limitations of existing approaches is essential for developing new models that can address the challenges of divergence and ensure reliable system behavior in high-throughput environments. This leads to the need for a more nuanced understanding of consistency in distributed architectures, which will be explored in the next section.

III. FUNDAMENTALS OF CONSISTENCY IN DISTRIBUTED ARCHITECTURES

Consistency in distributed systems is a multifaceted concept that encompasses multiple dimensions, including data integrity, system state alignment, and temporal coherence. Unlike centralized systems, where consistency can be enforced through atomic transactions, distributed systems must operate under conditions where strict consistency is often infeasible.

One of the foundational principles in distributed systems is the trade-off between consistency, availability, and partition tolerance. In practice, most large-scale systems prioritize availability and scalability, adopting eventual consistency models. In such systems, updates propagate asynchronously, and the system converges toward a consistent state over time rather than maintaining strict synchronization at every moment.

While eventual consistency provides scalability benefits, it introduces challenges in ensuring that system behavior remains coherent during intermediate states. Different services may have different views of the system at any given time, leading to temporary inconsistencies that must be managed carefully.

The SAGA pattern is often used to address these challenges by decomposing distributed transactions into smaller steps, each with its own compensating action. This approach allows systems to maintain a degree of consistency without requiring centralized coordination. However, it relies on the assumption that compensating actions can effectively reverse the effects of previous steps.

Traditional SAGA implementations focus on execution: which step comes next, and how to compensate when something goes wrong. But they do not guarantee that all paths will lead to the same final outcome. Two executions of the same business process, under slightly different conditions, can end in different states.

This limitation highlights a critical gap in traditional consistency models. While they address how systems execute and recover from failures, they do not ensure convergence toward a consistent outcome. In other words, they manage process correctness but do not guarantee outcome consistency.

This distinction becomes particularly important in high-throughput, event-driven environments, where variability in execution paths is inevitable. Differences in timing, ordering, and failure conditions can lead to divergent system states, even when each individual step is executed correctly.

To address this challenge, it is necessary to move beyond step-based consistency models and consider outcome-based approaches. Instead of focusing solely on how processes are executed, systems must be designed to ensure that they ultimately reach a coherent and intended state.

This shift in perspective forms the basis for the Convergence-Oriented SAGA model, which will be introduced in the next section as a new approach to achieving consistency in distributed systems.

IV. THE SAGA PATTERN: PRINCIPLES AND LIMITATIONS

The SAGA pattern has become one of the most widely adopted approaches for managing distributed transactions in microservices and event-driven architectures. Its primary objective is to provide a scalable alternative to traditional distributed transactions, which rely on strict coordination protocols such as two-phase commit. By decomposing a large transaction into a sequence of smaller, independent steps, SAGA enables systems to maintain operational continuity even in the presence of partial failures.

In a typical SAGA implementation, each step represents a local transaction executed by a service. These steps are coordinated either through orchestration, where a central controller manages the sequence, or choreography, where services react to events and trigger subsequent actions. When a step fails, compensating actions are invoked to reverse the effects of previously completed steps, with the goal of restoring system consistency.

This model offers several advantages. It reduces the need for global locks, supports asynchronous execution, and aligns well with the decentralized nature of microservices architectures. It also enables systems to remain available under high load, as services can continue processing events independently without waiting for global coordination.

However, the practical limitations of the SAGA pattern become evident in complex, high-throughput environments. While SAGA provides a mechanism for handling failures, it does not guarantee that all execution paths will lead to a consistent or equivalent final state. This limitation is particularly pronounced when compensating actions cannot fully reverse the effects of previous operations.

But here is the problem: not everything can be undone.

In real-world systems, certain actions produce irreversible effects. Financial transactions may be settled, external systems may be invoked, and physical processes such as shipping may already be underway. In such cases, compensating actions can only approximate reversal, often introducing additional complexity and potential inconsistencies.

This limitation is compounded by the asynchronous nature of event-driven systems. Delays, retries, and out-of-order event processing can create conditions where compensating actions are triggered too late or in incorrect sequences. As a result, the system may temporarily or permanently deviate from its intended state. In reality, things are rarely that clean.

Another critical issue is the lack of outcome determinism. SAGA implementations typically define the sequence of steps and their corresponding compensations, but they do not enforce a consistent end state across different executions. Variations in timing, failure conditions, and system load can lead to different outcomes for the same business process. Two executions of the same business process, under slightly different conditions, can end in different states.

This variability introduces a fundamental challenge in ensuring system reliability. While each individual step may be executed correctly, the overall system may exhibit divergent behavior due to the interaction of independent processes. This divergence is not

always immediately visible, making it difficult to detect and correct.

Another limitation of the SAGA pattern lies in its focus on execution flow. Traditional implementations emphasize the sequence of operations and the mechanisms for compensating failures. However, they do not provide a mechanism for continuously evaluating whether the system is moving toward a coherent outcome.

As systems scale, this limitation becomes more pronounced. The number of possible execution paths increases, and the likelihood of divergence grows. Without a mechanism to guide the system toward a consistent state, compensations alone are insufficient to ensure reliability.

These limitations suggest that while the SAGA pattern remains a valuable tool for managing distributed transactions, it must be extended or reinterpreted to address the challenges of modern, high-throughput systems. Specifically, there is a need to shift from a focus on execution correctness to a focus on outcome consistency.

This shift forms the basis for the Convergence-Oriented SAGA model, which introduces a new perspective on consistency by emphasizing system convergence rather than strict execution paths. This model is explored in detail in the next section.

V. FAILURE MODES IN HIGH-THROUGHPUT EVENT-DRIVEN SYSTEMS

In high-throughput, event-driven environments, failures are not isolated incidents but integral aspects of system behavior. These systems operate under continuous load, processing large volumes of events across distributed services. As a result, they are inherently subject to conditions such as network latency, partial failures, and asynchronous execution.

One of the most common failure modes is event reordering. Due to differences in processing speed and network delays, events may arrive at services in a different order than they were generated. This can lead to inconsistent state updates, particularly when operations depend on a specific sequence of events.

Another prevalent issue is event duplication, often caused by retries or at-least-once delivery semantics. While retries are necessary for ensuring reliability, they can result in the same operation being executed multiple times if idempotency is not properly enforced. A retry may trigger the same step multiple times.

Temporal delays also play a significant role in failure dynamics. Services may process events at different rates, leading to situations where some components advance while others lag behind. This creates temporal inconsistencies, where different parts of the system reflect different stages of a process. Delays between services can cause actions to arrive out of order.

In addition to these technical issues, partial failures introduce further complexity. A service may fail to process an event while others continue operating, leading to incomplete or inconsistent system states. These failures may not be immediately visible, allowing inconsistencies to propagate through the system. Over time, these factors create a system where different parts drift away from each other, even though each individual service behaves correctly.

This observation underscores a critical challenge: distributed systems can exhibit correct local behavior while producing incorrect global outcomes. Each service may fulfill its responsibilities accurately, yet the system as a whole may fail to achieve a coherent state. The system does not just fail—it diverges.

This concept of divergence is central to understanding failure in modern distributed systems. Unlike traditional failures, which are often abrupt and easily identifiable, divergence is gradual and cumulative. It arises from the interaction of multiple small inconsistencies that, over time, lead to significant deviations from the intended system state.

Another important failure mode is compensation failure. In SAGA-based systems, compensating actions are used to restore consistency when failures

occur. However, these actions may themselves fail, be delayed, or introduce additional inconsistencies. This creates cascading effects, where attempts to correct the system may inadvertently exacerbate the problem.

These failure modes highlight the limitations of traditional approaches to fault tolerance. Mechanisms such as retries, compensations, and local error handling address individual failures but do not ensure that the system converges toward a consistent state.

In high-throughput environments, where failures and inconsistencies are inevitable, fault tolerance must be redefined. Rather than focusing solely on preventing or correcting individual failures, systems must be designed to manage and reconcile inconsistencies over time. This perspective leads to the concept of convergence, where the system continuously adjusts its state to align with a desired outcome. The Convergence-Oriented SAGA model builds on this idea, providing a framework for achieving consistency through ongoing reconciliation rather than strict execution control. The next section introduces this model in detail, outlining its principles and architectural implications.

VI. CONVERGENCE-ORIENTED SAGA MODEL (CORE CONTRIBUTION)

The limitations of traditional SAGA implementations and the failure modes observed in high-throughput, event-driven systems point to a fundamental need for rethinking how consistency is achieved. Rather than attempting to enforce correctness through predefined execution paths and compensating actions alone, distributed systems must be designed to ensure that they converge toward a coherent outcome over time. To address this, we need to shift how we think about consistency. Instead of focusing on steps, we can focus on outcomes. This is the idea behind a Convergence-Oriented SAGA.

In a Convergence-Oriented SAGA model, the system does not rely on a fixed sequence of actions to guarantee consistency. Instead, it defines a desired final state and continuously evaluates its current state against this target. The primary responsibility of the system is not to execute a specific path, but to minimize the gap between the current state and the intended outcome.

In this model, the system defines a desired final state rather than a fixed sequence of actions. For an order, this might mean that payment is completed, inventory is reserved, and shipment is initiated. The system's responsibility is not to follow a strict path, but to continuously move toward this state.

This represents a conceptual shift from process-oriented consistency to state-oriented convergence. Traditional SAGA models focus on ensuring that each step is executed correctly and compensated when necessary. In contrast, the convergence-oriented model focuses on ensuring that, regardless of the execution path, the system ultimately reaches the correct state.

At the architectural level, this model introduces a continuous reconciliation loop. The system repeatedly observes its current state, compares it with the desired state, and applies corrective actions to reduce discrepancies. These corrective actions may include retries, compensations, or forward-progress operations, depending on the nature of the inconsistency.

Each service contributes to part of the overall state. When something goes wrong, the goal is not necessarily to undo previous actions, but to rebalance the system. If a payment is processed twice, the system issues a refund. If inventory is not reserved, it attempts to reserve it again. The focus is always on reducing the gap between the current state and the desired state.

This approach fundamentally changes the role of compensating actions. Instead of being used exclusively to reverse previous steps, compensations become part of a broader strategy for restoring balance. In some cases, moving forward (e.g., issuing a refund) may be more effective than attempting to revert past actions.

This creates a different kind of behavior. Instead of a linear sequence of steps, the system becomes a loop of reconciliation. It observes its current state,

compares it to the target, and applies whatever actions are needed to bring them closer together.

The reconciliation loop introduces a dynamic and adaptive execution model. Failures, retries, and delays are no longer treated as exceptional conditions but as normal aspects of system operation. The system is designed to tolerate these variations and adjust accordingly, ensuring that they do not lead to permanent inconsistencies. Failures, retries, and delays are no longer exceptional—they are part of the normal process.

A key advantage of this model is its ability to handle divergence. While traditional SAGA implementations may struggle to ensure consistent outcomes across different execution paths, the convergence-oriented approach actively drives the system toward alignment.

Because the system is continuously correcting itself, temporary inconsistencies do not accumulate into permanent errors. Even if parts of the system fall behind or behave unpredictably, the overall process still converges toward the intended outcome.

This property is particularly valuable in high-throughput environments, where variability in execution is inevitable. By focusing on convergence rather than strict execution order, the system can maintain consistency even under conditions of uncertainty.

From a design perspective, implementing a Convergence-Oriented SAGA requires several key capabilities:

- Representation of desired system state
- Continuous state observation and comparison mechanisms
- Support for idempotent and repeatable actions
- Mechanisms for forward correction and reconciliation

These capabilities enable the system to operate as a self-correcting entity, continuously adjusting its behavior to achieve the desired outcome. Of course, this requires a different way of thinking about compensation, idempotency, and state management.

The implications of this shift extend beyond implementation details. It requires rethinking how systems are designed, monitored, and operated. Engineers must move away from viewing workflows as fixed sequences and instead design systems that can adapt dynamically to changing conditions.

Despite these challenges, the benefits of convergence-oriented design are significant. It provides a more robust foundation for achieving consistency in distributed systems, particularly in environments where strict control over execution is not feasible. In high-throughput, event-driven environments, perfect execution is not realistic. But convergence is.

This statement encapsulates the core principle of the model. Rather than striving for perfect execution—which is often unattainable in distributed systems—the focus shifts to ensuring that the system can recover, adapt, and ultimately reach the correct state.

The architectural implications of this model, particularly in terms of state reconciliation and system design, are explored in the next section.

VII. STATE RECONCILIATION AND SYSTEM CONVERGENCE

State reconciliation is the operational mechanism through which convergence is achieved in distributed systems. It provides the processes and structures required to continuously align the current system state with the desired outcome.

In a convergence-oriented architecture, reconciliation is not a one-time corrective action but an ongoing process. The system maintains a representation of its current state and compares it against the desired state at regular intervals or in response to events. Discrepancies between these states trigger corrective actions designed to reduce divergence. Systems must track not just events, but the current state of the world.

This requirement marks a departure from traditional event-driven designs, which often focus primarily on event streams. While events remain essential for communication and coordination, they must be

complemented by state representations that capture the overall system condition.

Reconciliation mechanisms operate across multiple dimensions, including data consistency, process completion, and temporal alignment. For example, a reconciliation process may identify that a payment has been completed but inventory has not been reserved. In response, it can trigger actions to complete the missing step or adjust the system state accordingly.

A critical aspect of reconciliation is the design of corrective actions. These actions must be safe to execute multiple times and must not introduce additional inconsistencies. This requirement leads to the need for idempotent operations and carefully designed state transitions.

Another important consideration is the frequency and granularity of reconciliation. Frequent reconciliation can improve consistency but may introduce additional computational overhead. Coarse-grained reconciliation reduces overhead but may allow inconsistencies to persist longer. Balancing these factors is essential for achieving efficient and effective convergence.

Observability plays a key role in supporting reconciliation. Systems must provide visibility into both current and desired states, as well as the actions being taken to reconcile them. This enables operators to understand system behavior and intervene when necessary.

Ultimately, state reconciliation transforms fault tolerance from a reactive process into a proactive and continuous capability. Instead of responding to failures after they occur, the system continuously monitors and adjusts itself to maintain alignment with its intended state.

The next section examines the design implications of this approach, particularly in relation to idempotency, compensation, and state management.

VIII. IDEMPOTENCY, COMPENSATION, AND STATE MANAGEMENT

The effectiveness of a Convergence-Oriented SAGA model depends critically on how systems handle idempotency, compensation, and state representation. These elements, while present in traditional distributed architectures, take on a fundamentally different role in convergence-based systems. Instead of serving as auxiliary mechanisms for failure recovery, they become central design principles that enable continuous reconciliation and reliable convergence.

Idempotency is a foundational requirement in this context. In high-throughput, event-driven environments, operations may be executed multiple times due to retries, duplicate events, or delayed processing. Without idempotency, repeated execution can lead to inconsistent system states and unintended side effects. In convergence-oriented systems, where actions may be re-applied as part of reconciliation, idempotency is not optional—it is essential.

An operation is idempotent if executing it multiple times produces the same effect as executing it once. Designing idempotent operations requires careful consideration of how state changes are applied. This often involves the use of unique identifiers, versioning mechanisms, and conditional updates that ensure repeated actions do not introduce inconsistencies.

In traditional SAGA implementations, compensation is primarily used to reverse the effects of failed steps. However, in a convergence-oriented model, compensation is reinterpreted as one of several mechanisms for restoring alignment between current and desired states. Rather than strictly undoing previous actions, compensations may also involve forward corrections that adjust the system toward the intended outcome.

For example, if a duplicate payment is detected, the appropriate corrective action may be issuing a refund rather than attempting to reverse the original transaction. This approach recognizes that certain actions cannot be undone and instead focuses on achieving a balanced and consistent state.

This shift requires a broader definition of compensation, one that includes both reversal and correction. Compensation strategies must be designed to handle partial failures, irreversible actions, and asynchronous interactions. They must also be safe to execute multiple times, reinforcing the importance of idempotency.

State management is another critical component of convergence-oriented systems. Traditional event-driven architectures often emphasize event logs as the primary source of truth. While event logs are valuable for tracking system behavior, they do not inherently provide a complete or current view of system state. Systems must track not just events, but the current state of the world.

This requirement necessitates the introduction of state representations that capture the aggregate condition of the system. These representations may be implemented through state stores, materialized views, or distributed data structures that reflect the current status of each process.

State management in convergence-oriented systems involves maintaining both the current state and the desired state. The reconciliation process continuously compares these states and identifies discrepancies. Effective state management must therefore support efficient querying, updating, and synchronization across distributed components.

Another important aspect is state granularity. Fine-grained state representations provide detailed insights into system behavior but may increase complexity and overhead. Coarse-grained representations simplify management but may obscure important details. Selecting the appropriate level of granularity is essential for balancing performance and observability.

Temporal considerations also play a role in state management. In distributed systems, state changes occur over time and may not be immediately visible across all components. Managing temporal inconsistencies requires mechanisms for tracking event ordering, handling delayed updates, and ensuring eventual convergence.

The integration of idempotency, compensation, and state management creates a cohesive framework for supporting convergence. These elements work together to ensure that actions can be safely repeated, inconsistencies can be corrected, and system state can be accurately represented and reconciled.

From an architectural perspective, these capabilities must be embedded into system design from the outset. Retrofitting idempotency or state management into existing systems can be challenging and may introduce additional complexity. Designing form convergence requires a holistic approach that considers how services interact, how state is represented, and how inconsistencies are resolved.

Ultimately, the goal is to create systems that are resilient not because they avoid failures, but because they can adapt to them. By ensuring that actions are repeatable, compensations are effective, and state is well-managed, convergence-oriented architectures provide a robust foundation for achieving consistency in distributed environments.

The next section explores how observability must evolve to support this model, particularly in terms of understanding system convergence and detecting divergence.

IX. OBSERVABILITY IN CONVERGENCE-BASED SYSTEMS

Observability in traditional distributed systems is primarily concerned with understanding execution: which services were invoked, what events were processed, and how requests propagated across the system. Logs, metrics, and traces are used to reconstruct system behavior and diagnose failures. While these tools remain essential, they are not sufficient in convergence-oriented architectures, where the central concern shifts from execution flow to state alignment.

In convergence-based systems, observability must answer a fundamentally different question: How far is the system from its intended state? This requires visibility not only into what has happened, but into what should have happened and whether the system is progressing toward that outcome. Observability

must evolve to show how far the system is from its intended outcome, not just what steps have been executed.

This shift introduces the concept of state-centric observability. Instead of focusing solely on event traces, the system exposes its current state, desired state, and the gap between them. This gap becomes a primary indicator of system health and correctness.

A convergence-oriented observability model typically includes the following components:

- Current State View: A representation of the system's present condition across
- distributed services
- Desired State Definition: A formal description of the expected final state for
- each process
- State Delta (Gap): The measurable difference between current and desired
- states
- Reconciliation Actions: The operations currently being applied to reduce this gap

By continuously monitoring these elements, the system can provide real-time insight into whether it is converging or diverging.

Another important aspect is temporal observability. In event-driven systems, delays and asynchronous execution can obscure the relationship between cause and effect. Convergence-based observability addresses this by tracking how state evolves over time rather than relying solely on event sequences. This enables the system to detect whether progress is being made toward the desired outcome, even in the presence of delays or partial failures.

Traditional tracing systems attempt to reconstruct execution paths after the fact. In contrast, convergence-oriented observability focuses on interpreting the present state. This reduces the need for complex trace reconstruction and allows operators to identify inconsistencies more directly.

For example, instead of tracing a failed workflow across multiple services, an operator can observe that a process is in a partially completed state—such as payment completed but inventory not reserved—and immediately understand the nature of the discrepancy. Debugging becomes less about reconstructing the past and more about interpreting the present.

This approach significantly reduces cognitive complexity in large-scale systems. As the number of services and interactions increases, reconstructing event histories becomes increasingly difficult. State-centric observability provides a more intuitive and scalable way to understand system behavior.

Another key capability is divergence detection. Since convergence is the primary goal, the system must be able to identify when it is moving away from the desired state. Divergence may occur due to repeated failures, inconsistent data, or unexpected interactions between services. By monitoring state deltas over time, the system can detect when discrepancies are increasing rather than decreasing, signaling a need for intervention.

Observability also plays a critical role in automated reconciliation. The system can use observed state discrepancies to trigger corrective actions without human intervention. For example, if a required state transition has not occurred within a certain timeframe, the system can initiate retries or alternative actions to restore alignment.

Visualization tools are particularly important in this context. Dashboards that display state progression, convergence metrics, and reconciliation actions provide operators with actionable insights. These tools must present complex distributed behavior in a simplified and interpretable form, enabling rapid decision-making.

From an architectural perspective, implementing convergence-based observability requires integrating state tracking mechanisms with monitoring infrastructure. This may involve combining event streams with state stores, enabling real-time updates and queries. It also requires defining clear models for representing desired states and measuring convergence.

Ultimately, observability in convergence-oriented systems transforms from a diagnostic tool into a core control mechanism. It enables systems not only to detect issues but to actively guide themselves toward consistent outcomes.

The practical implications of this approach are further illustrated in the next section through case-based scenarios that demonstrate convergence in action.

X. CASE-BASED ANALYSIS OF CONVERGENCE IN DISTRIBUTED EVENT-DRIVEN SYSTEMS

To evaluate the implications of convergence-oriented design, it is useful to analyze distributed workflows under different architectural models. Rather than relying on anecdotal descriptions, this section examines representative execution patterns and their resulting system behaviors, focusing on consistency, fault tolerance, and outcome determinism.

10.1 Execution-Based SAGA Model

Consider a distributed transactional workflow consisting of multiple dependent operations, such as payment processing, inventory reservation, and shipment initiation. In a traditional SAGA implementation, each operation is executed as an independent local transaction, with compensating actions defined to handle failures. Under ideal conditions, this model provides a structured approach to managing distributed consistency. However, in high-throughput environments, execution is subject to variability in timing, ordering, and failure conditions. When a failure occurs—for example, inventory reservation fails after payment has been processed—the system initiates compensating actions to revert previously completed steps. While this mechanism is theoretically sound, its effectiveness depends on the assumption that all operations are reversible and that compensations can restore the system to a consistent baseline. In practice, this assumption does not hold universally. Certain operations introduce irreversible side effects or interact with external systems beyond transactional control. As a result, the execution-based SAGA model ensures process-level correctness, but does not guarantee outcome-level consistency.

10.2 Divergence Under Asynchronous Execution

In event-driven systems operating at scale, multiple instances of the same workflow may execute concurrently. These executions are influenced by non-deterministic factors such as network latency, event reordering, and retry mechanisms. Under these conditions, the system may exhibit divergence, where different components reflect inconsistent states of the same logical process. Over time, these factors create a system where different parts drift away from each other, even though each individual service behaves correctly. This phenomenon arises because distributed systems lack a global synchronization point. Each service operates based on local knowledge and event streams, leading to discrepancies in state representation across the system.

From a formal perspective, let Si(t)S_i(t)Si(t) represent the state of service iii at time ttt. Divergence occurs when:

∃i,j:Si(t)≠Sj(t)with respect to the same process instance\exists i, j : S_i(t) \neq S_j(t) \quad \text{with respect to the same process instance} ∃i,j:Si(t)=Sj(t)with respect to the same process instance.

This divergence may persist or increase over time if corrective mechanisms are not applied effectively. The system does not just fail—it diverges.

Unlike traditional failures, divergence is not necessarily detectable through standard error signals. Instead, it manifests as gradual misalignment between system components, making it more difficult to identify and resolve.

10.3 Limitations of Compensation-Based Recovery

Traditional SAGA implementations rely on compensating actions to restore consistency. However, compensation mechanisms are inherently reactive and limited in scope.

They operate under three implicit assumptions:

1. All actions are reversible
2. Compensation can be executed reliably
3. Execution paths converge after rollback

In real-world systems, these assumptions are frequently violated. Compensation may fail, be delayed, or introduce additional inconsistencies. Moreover, reversing an action does not always restore the system to its original state due to side effects and temporal dependencies.

This leads to a critical limitation: compensation-based recovery addresses local inconsistencies, but does not ensure global convergence.

10.4 Convergence-Oriented SAGA Model

To address these limitations, consider a convergence-oriented approach in which the system is defined in terms of a desired final state rather than a fixed sequence of operations. The focus is always on reducing the gap between the current state and the desired state.

Let:

- S(t)S(t)S(t): current system state 22
- S∗S^*S∗: desired state

The objective of the system becomes:

limt→∞Δ(S(t), S∗) →0\lim_ {t \to \infty} \Delta(S(t), S^*) \to 0t→∞limΔ(S(t), S∗) →0

represents a measure of divergence between current and desired states.

In this model, execution paths are not strictly enforced. Instead, the system continuously evaluates its state and applies corrective actions to reduce divergence. Instead of a linear sequence of steps, the system becomes a loop of reconciliation.

This loop introduces a feedback-driven execution model in which:

- State is observed
- Deviations are detected
- Actions are applied to restore alignment

Unlike traditional SAGA implementations, convergence-oriented systems do not rely solely on compensations. They may apply forward corrections, such as issuing refunds or re-executing missing steps, depending on the nature of the discrepancy.

10.5 Continuous Reconciliation Dynamics

The convergence-oriented model can be interpreted as a dynamic system with feedback control. At each iteration:

S(t+1)=S(t)+A(t)S(t+1) = S(t) + A(t)S(t+1)=S(t)+A(t) where A(t)A(t)A(t) represents corrective actions applied based on observed divergence.

This formulation highlights several key properties:

- Robustness to failure: Temporary inconsistencies do not accumulate
- Tolerance to non-determinism: Variations in execution paths are absorbed
- Outcome determinism: Final state converges toward S∗S^*S∗

Because the system is continuously correcting itself, temporary inconsistencies do not accumulate into permanent errors. Even if parts of the system fall behind or behave unpredictably, the overall process still converges toward the intended outcome.

This behavior contrasts with execution-based models, where inconsistencies may persist or amplify due to incomplete compensations.

10.6 Implications for High-Throughput Systems

In high-throughput environments, strict execution guarantees are often impractical due to scale and variability. Convergence-oriented design provides a more realistic framework by acknowledging that:

- Failures are inevitable
- Events are not perfectly ordered
- Execution paths cannot be fully controlled

In high-throughput, event-driven environments, perfect execution is not realistic. But convergence is. By shifting the focus from execution correctness to outcome convergence, systems can achieve higher levels of fault tolerance and consistency under real-world conditions.

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1715957

XI. TRADE-OFFS AND SYSTEM CONSTRAINTS

While the Convergence-Oriented SAGA model provides a more robust framework for achieving consistency in distributed systems, it introduces a set of architectural trade-offs that must be carefully managed. These trade-offs are not merely implementation details; they reflect fundamental tensions between control and adaptability, simplicity and robustness, and performance and reliability.

One of the primary implications of convergence-oriented design is the shift from execution control to state management. Traditional SAGA implementations rely on predefined execution paths and explicit coordination logic. In contrast, convergence-based systems require continuous observation and evaluation of system state. This introduces additional complexity, as the system must maintain both a representation of its current state and a definition of its desired outcome. Managing this dual-state model at scale requires efficient data structures, consistent state synchronization mechanisms, and careful handling of temporal inconsistencies.

This architectural shift also impacts system performance. Continuous reconciliation introduces additional processing overhead, as the system must repeatedly compare current and desired states and apply corrective actions. In high-throughput environments, where thousands or millions of events are processed concurrently, this overhead can become significant if not carefully optimized. The design must therefore balance the frequency and granularity of reconciliation with system performance requirements.

At the same time, convergence-oriented systems rely heavily on idempotent and repeatable operations. Actions may be executed multiple times as part of the reconciliation process, and therefore must be designed to avoid unintended side effects. This requirement extends across all services, affecting how APIs are designed, how state transitions are implemented, and how data is stored and updated. Achieving true idempotency in complex systems is non-trivial and often requires additional metadata, versioning strategies, or conditional logic.

Another important consideration is the handling of irreversible operations. As previously discussed, certain actions cannot be undone once executed. Convergence-oriented systems address this by introducing forward corrections rather than relying solely on compensation. However, this approach requires careful design to ensure that corrective actions do not introduce new inconsistencies. For example, issuing refunds to correct duplicate payments must be coordinated with financial systems, accounting processes, and external constraints.

The reliance on state representations also introduces challenges in terms of consistency and synchronization. In distributed environments, different services may have different views of the system state at any given time. Ensuring that reconciliation decisions are based on sufficiently accurate and up-to-date information is critical. This may require mechanisms such as state versioning, eventual consistency models, or consensus-based updates, each of which introduces its own trade-offs.

Observability requirements also become more demanding. In convergence-based systems, it is not sufficient to monitor individual services or event flows. The system must provide visibility into its current state, desired state, and the gap between them. This requires more sophisticated monitoring infrastructure, capable of aggregating and interpreting distributed state information in real time.

Despite these challenges, the convergence-oriented approach offers a significant advantage in terms of fault tolerance. By continuously correcting discrepancies, the system reduces the risk of permanent inconsistencies. Temporary failures, delays, and variations in execution are absorbed into the reconciliation process, allowing the system to recover without requiring strict coordination or rollback mechanisms.

However, this benefit comes at the cost of increased architectural and operational complexity. Designing, implementing, and maintaining convergence-oriented systems requires a deeper understanding of

distributed system behavior and a more sophisticated approach to system design. Organizations must also adapt their development and operational practices to support continuous reconciliation and state-centric thinking.

Ultimately, the trade-offs associated with convergence-oriented design reflect a broader reality of distributed systems: achieving reliability at scale requires accepting and managing complexity rather than eliminating it. The goal is not to simplify the system by enforcing strict control, but to design it in a way that can adapt to variability and maintain coherence over time.

XII. FUTURE DIRECTIONS

The shift toward convergence-oriented design reflects a broader evolution in distributed systems, where resilience is achieved not through strict control but through adaptive behavior. As systems continue to scale and operate under increasingly dynamic conditions, architectural models must evolve to support greater flexibility, autonomy, and fault tolerance.

One of the most promising directions is the integration of convergence principles with real-time stream processing systems. Modern event streaming platforms enable continuous data processing at scale, providing a natural foundation for implementing reconciliation loops. By combining stream processing with state-aware logic, systems can achieve near real-time convergence, reducing the duration and impact of inconsistencies.

Another emerging trend is the use of intelligent monitoring and adaptive control mechanisms. Machine learning techniques can be applied to detect patterns of divergence, predict potential inconsistencies, and recommend or automate corrective actions. This introduces the possibility of self-healing systems that not only react to failures but anticipate and prevent them.

The concept of convergence can also be extended to multi-system environments, where distributed processes span organizational boundaries. In such scenarios, ensuring consistency becomes even more challenging, as systems may operate under different constraints and governance models. Developing standardized approaches to representing desired states and measuring convergence across systems will be an important area of research.

From a design perspective, future systems are likely to place greater emphasis on declarative models, where desired outcomes are specified explicitly and systems determine how to achieve them. This aligns closely with convergence-oriented thinking, as it shifts the focus from execution logic to outcome definition.

At the same time, advances in observability will play a crucial role in supporting these architectures. Visualization tools and monitoring systems will need to evolve to provide clear insights into system convergence, enabling operators to understand and guide system behavior effectively.

Despite these advancements, challenges remain. Managing the complexity of convergence-oriented systems, ensuring the reliability of state representations, and balancing automation with human oversight will require ongoing attention. Additionally, the adoption of such models will depend on organizational readiness and the ability to integrate new architectural principles into existing systems.

In the end, fault tolerance is not about preventing things from going wrong. It is about ensuring that, no matter what happens, the system finds its way back to the right state. This principle encapsulates the core idea of convergence-oriented design. By focusing on outcomes rather than execution paths, systems can achieve a higher level of resilience and consistency, even in the presence of uncertainty and variability.

XIII. CONCLUSION

The design of fault-tolerant, event-driven systems requires a fundamental rethinking of how consistency is achieved in distributed environments. Traditional approaches, including execution-based SAGA implementations, provide mechanisms for managing distributed transactions but fall short in ensuring consistent outcomes under real-world conditions.

This paper has examined the limitations of these approaches, particularly in high-throughput systems where failures, delays, and non-deterministic behavior are inherent. It has introduced the concept of a Convergence-Oriented SAGA model, which shifts the focus from execution correctness to outcome convergence.

By defining desired system states and continuously reconciling current states against them, convergence-oriented systems provide a more robust framework for achieving consistency. This approach acknowledges the realities of distributed systems and leverages them as part of the solution, rather than attempting to eliminate them.

The analysis has demonstrated that convergence-oriented design enhances fault tolerance by absorbing variability and preventing the accumulation of inconsistencies. It also introduces new requirements for state management, idempotency, and observability, which must be carefully addressed in system design.

Ultimately, the key contribution of this work lies in reframing consistency as a dynamic process rather than a static guarantee. By focusing on where the system needs to end up, rather than how it gets there, it is possible to design architectures that remain reliable even under complex and unpredictable conditions.

This perspective opens new directions for research and practice in distributed systems, providing a foundation for building resilient, scalable, and consistent architectures in the era of high-throughput event-driven computing.

REFERENCES

[1] Birman, K. P. (2012). Guide to reliable distributed systems: Building high-assurance applications and cloud-hosted services. Springer.

[2] Brewer, E. A. (2012). CAP twelve years later: How the “rules” have changed. Computer, 45(2), 23–29. https://doi.org/10.1109/MC.2012.37

[3] Garcia-Molina, H., & Salem, K. (1987). Sagas. ACM SIGMOD Record, 16(3), 249–259. https://doi.org/10.1145/38713.38742

[4] Helland, P. (2007). Life beyond distributed transactions: An apostate’s opinion. CIDR Conference.

[5] Kleppmann, M. (2017). Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. O’Reilly Media.

[6] Kreps, J. (2011). Kafka: A distributed messaging system for log processing. NetDB Workshop.

[7] Newman, S. (2021). Building microservices: Designing fine-grained systems (2nd ed.). O’Reilly Media.

[8] Nygard, M. T. (2018). Release it!: Design and deploy production-ready software (2nd ed.). Pragmatic Bookshelf.

[9] Pat Helland, (2016). Immutability changes everything. Communications of the ACM, 59(1), 54–60. https://doi.org/10.1145/2845385

[10] Pautasso, C., Zimmermann, O., & Leymann, F. (2017). Microservices in practice, part 1: Reality check and service design. IEEE Software, 34(1), 91–98. https://doi.org/10.1109/MS.2017.24

[11] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys, 22(4), 299–319. https://doi.org/10.1145/98163.98167 30

[12] Tanenbaum, A. S., & van Steen, M. (2017). Distributed systems: Principles and paradigms (2nd ed.). Pearson.

[13] Vogels, W. (2009). Eventually consistent. Communications of the ACM, 52(1), 40–44. https://doi.org/10.1145/1435417.1435432