10.48047/jocaaa.2026.35.03.24

Event-Driven Real-Time Salesforce Architectures at Scale: A Reference Model for High-Volume Enterprise Systems

Aditya Pothukuchi
Brenntag North America Inc., USA

Abstract

Enterprise customer relationship management platforms have evolved from structured sales repositories into mission-critical operational hubs processing vast volumes of customer interactions across multiple integrated systems. Conventional synchronous integration architectures, including batch-based data exchange, outbound messaging, and request-response API invocations, impose structural constraints that become operationally consequential at enterprise scale—manifesting as latency accumulation, throughput saturation at platform governor limits, and cascading failure propagation across tightly coupled integration chains. These limitations directly conflict with the real-time engagement expectations that customers now hold toward the enterprises they interact with, as evidenced by documented shifts in consumer brand loyalty, personalization expectations, and data-sharing behavior. Event-driven architecture has emerged as a structurally appropriate resolution to this class of integration challenge, replacing synchronous coupling with a durable, decoupled event backbone through which producers and consumers interact asynchronously and independently. This article proposes a comprehensive reference model for implementing event-driven principles within the governance constraints of large-scale Salesforce deployments, introducing a canonical event schema for cross-system interoperability, adapting event sourcing and command-query responsibility segregation patterns to CRM domain structures, and embedding idempotent processing, distributed observability, and governance controls throughout the architectural fabric. Empirical validation conducted within a global enterprise deployment confirms substantial improvements in integration latency, system throughput, fault tolerance, and deployment agility relative to the incumbent synchronous architecture. Peak load behavior, zero-downtime upgrade capability, and disaster recovery performance through asynchronous decoupling are each validated against baseline measurements. The outcomes align with theoretical predictions from distributed systems literature and extend existing benchmarks into the SaaS-constrained CRM context for the first time. The implications of this article extend beyond Salesforce to any enterprise SaaS platform exposing native publish-subscribe or change data capture capabilities, offering a replicable and theoretically grounded blueprint for real-time CRM modernization across regulated and high-volume industries.

Keywords: Event-Driven Architecture, Salesforce Platform Integration, Command-Query Responsibility Segregation, Canonical Event Schema, Enterprise SaaS Scalability

1. Introduction

The customer relationship management landscape has been experiencing a radical structural change due to the collision of the increasing consumer demands, the growth of digital channels, and the speed at which enterprises are adopting technology. What used to be a platform whose main purpose was to display the sales pipeline has become a crucial operational layer that links customer-facing processes in the areas of service delivery, commerce, marketing, and field operations. Studies on related customer behavior demonstrate that the expectations that customers now have of the enterprises they interact with

10.48047/jocaaa.2026.35.03.24

have changed significantly—consumers are more and more reconsidering their priorities as 82% of them use the cost of living as a key driver of shifting expectations, and 34% of them refer to technological progress as a key factor that influences such expectations [1]. The repercussions of falling short of those expectations are provably behavioral: out of the number of consumers who had changed brands over the course of the last year, 72% did so because they thought they should get a better deal, 55% and 44% because of product quality grievances, and 41% because the product was out of stock or because of a limited selection [1].

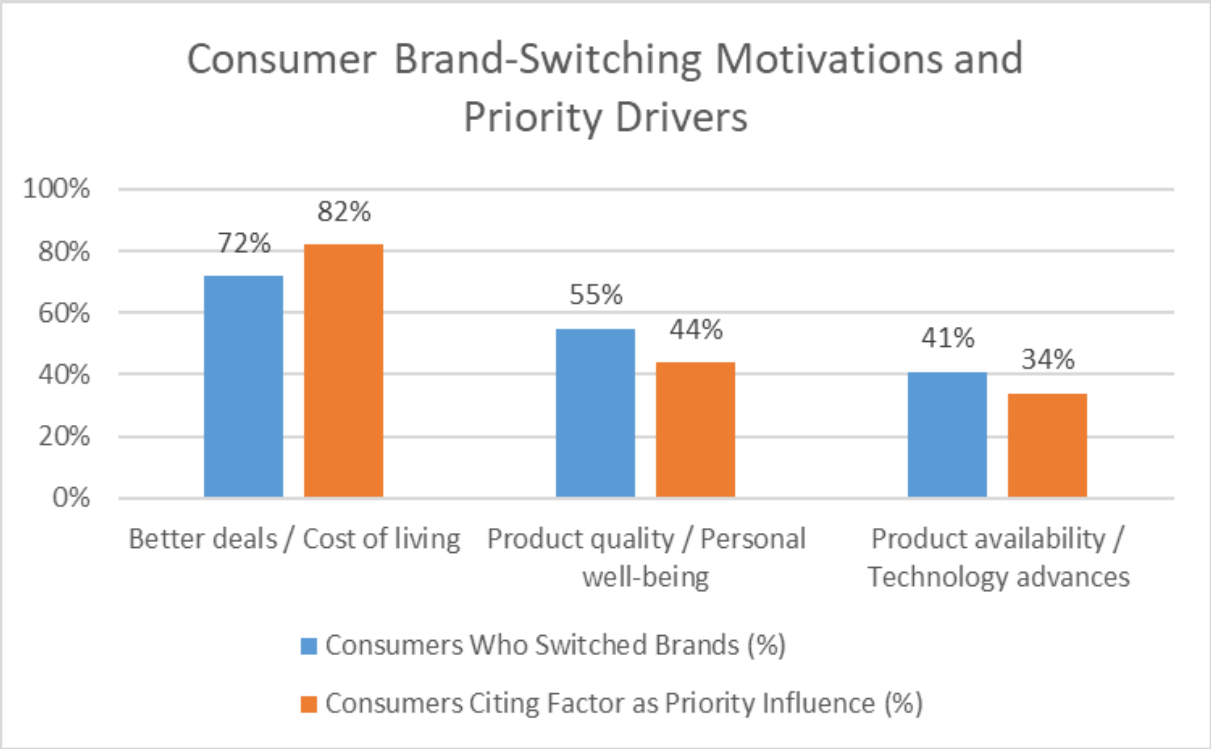

Fig 1: Consumer Brand-Switching Motivations and Priority Drivers. [1]

In this respect, the personalization disparity between customer expectation and the current provision by enterprises can be considered one of the most significant operational problems of CRM-dependent companies. According to the survey data, 65% of clients believe that companies should orient to their changing needs and preferences, but 64% of customers state that most companies use this as a number instead of as individuals with specific needs [1]. This detachment is also made clear with brand adaptation information, which indicates that only 16% of consumers think their favorite brands continuously tailor to their evolving needs, with the percentage dropping to 12% with the majority of brands, which indicates a systemic failure of personalization at the scale affecting even relationship-based consumer affinity [1]. Companies that have defined CRM integrations based on traditional synchronous and batch-based architectures are structurally bound in their capacity to bridge this divide since the latency between planned data sharing does not allow the real-time signal processing required to support adaptive and personalized engagement.

This is further enhanced by the data imperative. Even amongst customers who desire speed, 81% do connect their desire with the technology development, 75% with their readiness to give more personal

10.48047/jocaaa.2026.35.03.24

information, and 65% with their amount of financial expenditure, which created the clear customer-side contract in which the sharing of data is provided in exchange for more responsive service [1]. Similarly, 74 and 73 percent of the customers anticipate greater personalization when they submit additional data and when responding to technological advancements, respectively [1]. These numbers are all that constitute a value exchange model whereby the CRM platform should process and act on customer information in real time to respect the implicit contract that customers commit to by giving personal information. Request-response integrations run synchronously, and batch processing cycles cannot architecturally meet this contract at transaction volumes that are typical of large enterprise implementations, where the cycle between data ingestion and actionable insight is regularly measured in hours and not in milliseconds.

Enterprise architecture research has found event-driven architecture to be a paradigm through which this type of structural limitation can be dealt with. The fundamental concept of EDA—that the components of the system can only communicate with one another by generating and consuming discrete, unalterable records of state change but not by explicit procedural invocation the temporal and logical linkage of the synchronous integration. An event-based topology In an event-driven topology, one producing system issues an event to a common messaging infrastructure with no requirement based on the availability, identity, or speed of downstream consumers. Relevant event streams are subscribed to by consumers, and event processing occurs independently so that each component can scale, fail, and evolve without coordination with its integration partners. Research on enterprise architecture has confirmed that this decoupling model has much higher scalability, fault isolation, and flexibility in complex multi-systems, where the number of integration dependencies and inter-system communications volume increase in a manner that synchronous architecture cannot sustainably support [2]. The event-driven paradigm also brings a long-lasting record of all inter-domain state transitions, a feature that has a direct utility in the audit, compliance, and system recovery needs that are especially sharp in CRM-centric implementations working with sensitive customer and transactional information [2].

10.48047/jocaaa.2026.35.03.24

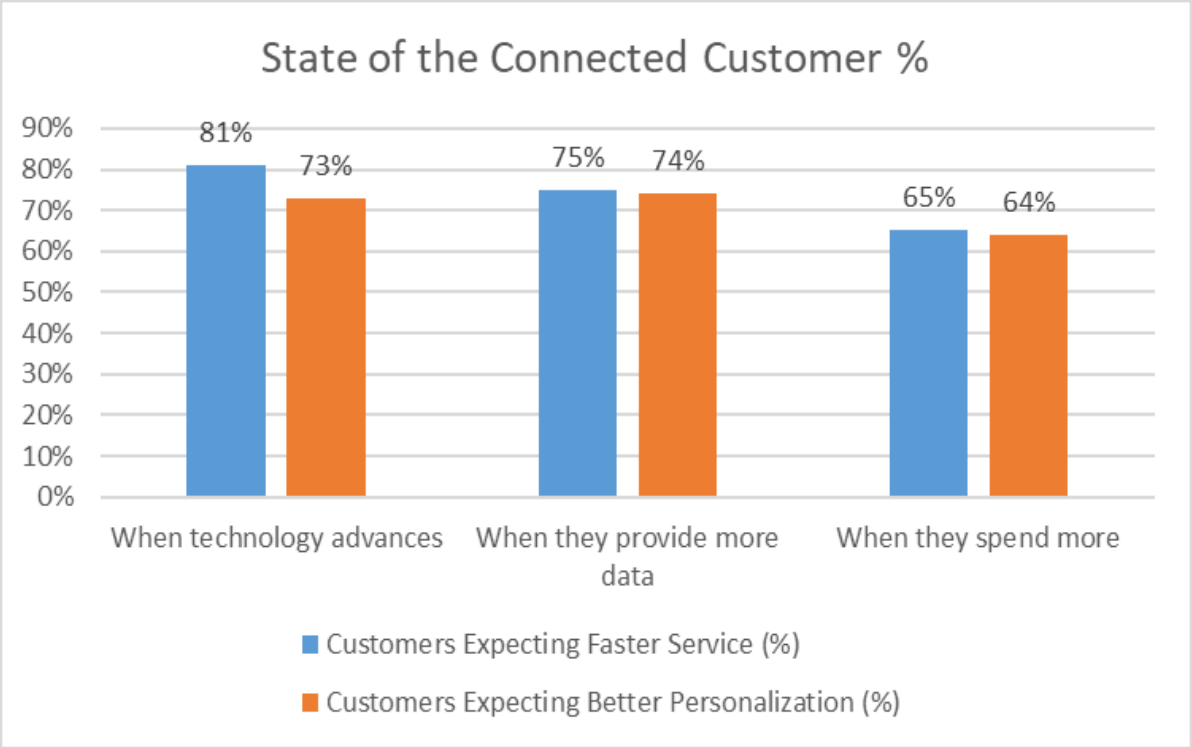

Fig. 2: Customer Expectations for Faster Service Based on Engagement Conditions. [1, 2]

Trust and data governance factors support the criticality of architectural modernization. It has been found out that 80 percent of the customers feel that their experiences are supposed to be better informed by the kind of data the companies are gathering, and 79 percent of customers observe that they are growing more protective of their personal data [1]. Moreover, 71 percent of the customers state that they would be more likely to entrust their personal data to a company when they have a clear understanding of their intended use, and 51 percent of the customers are convinced that the data is not being used by the majority of the companies in a manner that would benefit them [1]. These results prove that the business case of real-time CRM architecture cannot be made without a greater governance requirement: business organizations need not only to process customer information more quickly but also more effectively by designing data streams governed by a schema and event records (that can be audited) and data streams (that can be controlled) that can meet regulatory obligations, as well as customer demands of accountability. Although the theoretical and empirical rationale of event-driven integration is well-established, systematic directions on applying EDA within the governance features of low-code and SaaS-based architecture like Salesforce are underdeveloped, which is the main research gap addressed in this paper [2].

The paper suggests and justifies an all-encompassing reference design of planning and deploying event-driven Salesforce architecture on a large-scale basis. The model also proposes a canonical event schema to support cross-system interoperability, transforms event sourcing and command query responsibility segregation patterns to CRM enterprise domain structures, and implements observability and governance features to mission-critical enterprise workloads. Empirical validation comes out of a large-scale deployment of production processing of more than ten million daily records with benchmarks that measure the reduction in integration latency, fault tolerance, and deployment agility over a baseline synchronous architecture. The rest of the paper is organized in the following way: Section 2 discusses the

10.48047/jocaaa.2026.35.03.24

background literature and previous work related to the proposed topic; Section 3 provides the reference model in its entirety; Section 4 reports the results of the empirical evaluation and comparative performance analysis; and Section 5 makes the conclusions and proposes the ways to address the topic of the research in the enterprise.

2: Background and Related Work.

Three interdependent structural components comprise event-driven architecture, including producers, consumers, and an event broker, whose interaction determines the basic communication pattern of any EDA implementation. Producers are elements in a system that identify and emit discrete records of state change events, with no knowledge of what downstream systems will accept or respond to them. Consumers are unilaterally running elements that subscribe to the appropriate event streams and implement process logic when received, and they are no more conscious of who the producer is or what behavior they perform. The event broker, which is realized by Apache Kafka, AWS EventBridge, or Salesforce Platform Events technologies, is the distributed infrastructure, which accepts, durably stores, and directs events among producers and consumers. Apache Kafka is based on a distributed, partitioned commit log design and has emerged as the ubiquitous backbone technology of the enterprise, with support of high-throughput, fault-tolerant event streaming and configurable retention and consumer replay functionality. AWS EventBridge is a cloud-native extension of this model that offers serverless event routing, including schema registry and cross-account delivery. Platform Events within the Salesforce ecosystem also follow a publish-subscribe implementation of the CometD streaming protocol, allowing near-real-time message delivery between Salesforce components and those of external subscribers without polling. The existing body of research on the implementation of streaming platforms in enterprise distributed systems shows that partitioned log architectures offer the best trade-off between throughput, ordering properties, and durability to mission-critical workloads, making event backbone the base infrastructure layer on which higher-order EDA patterns are built [3].

The use of EDA in distributed systems of enterprises and cloud-native has been widely studied in the financial services, telecommunications, logistics, and e-commerce sectors. Experimental work in these domains has continually shown that event-based integration causes less inter-service coupling, better fault containment, and horizontal scaling, which synchronous architectures do not have the ability to replicate in the presence of variable load environments. Cloud-native applications based on containerized microservices have especially enjoyed the support of EDA in line with the principles of independent deployability and limited context of modern distributed system architecture. The studies of cloud-environment EDA implementations go further to provide the fact that more resilient failure recovery is possible when asynchronous event propagation is used, since the accumulation of consumer lag during partial outages can be recovered by replay without data loss, which is not a property of synchronous systems where failed calls must be explicitly orchestrated [4]. The maturity of EDA uptake in the realms gives a tested theoretical and empirical base on which CRM-specific applications can rely, but the governance limitations of SaaS present adjustment needs that are not met by the current frameworks formulated in infrastructurally unrestricted settings.

The Salesforce integration frameworks have been analyzed in the preceding discussion, which has identified the scaled limitations of the Salesforce platform native integration models that are well documented. Although able to react to record-level operations of the data manipulation language in near real time, apex triggers are limited to governor limits, including the number of callouts, DML statements

10.48047/jocaaa.2026.35.03.24

and processor execution time allocated to a transactional context—limits which become binding at enterprise transaction volumes. Outbound messaging, the older Salesforce workflow-based notification system, is a synchronous HTTP system with blocking retry capabilities that limit performance owing to load and have no buffering of consumer unavailability. Flexible, widely used REST and SOAP API integrations share the same limiting characteristics of synchronous invocation to bind integration throughput to API governor limit allocations, distribute consumer errors upstream to the calling processes, and offer no native support to ordering of events or exactly-once delivery. These constraints are jointly sufficient to determine that the current primitives of the platform, although sufficient to support moderate transactional workloads, are inadequate in structure to support the deployment of enterprise-scale systems demanding continuous and high-volume data transfer and exchange between and among a variety of systems with which the platform has been integrated [3].

The architectural designs best applicable to high-volume CRM integration are event sourcing, command-query responsibility segregation, and idempotent processing, which deal with different aspects of the scalability and consistency issues of synchronous Salesforce integration. Event sourcing substitutes state records with an append-only log of domain events, which is immutable and on which current state is computed through sequential projection, and which offers a full and reproducible audit trail that is especially useful in CRM settings with compliance implications. CQRS separates the write (optimized to provide transactional consistency) and the read (optimized to respond to queries) paths and thus allows the reporting and operational workloads to scale independently, as they would otherwise compete over common API bandwidth. Idempotent processing solves the at-least-once delivery property of most streaming platforms; that is, duplicate event delivery is allowed to yield identical state behavior, a property necessary to support CRM data integrity in distributed processing pipelines. Although all of these patterns have already been well theorized and proven in the context of cloud-native, their systematic application to the unique data structure, governor policies, and declarative limits of automation of Salesforce-centric deployments is an open contribution to the existing literature, and the lack of an integrated, SaaS-focused reference model that brings these patterns to a consistent operational model is the main research gap that this paper will address [4].

3: Event-Driven Salesforce Architecture Proposed Reference Model.

The reference model divides Salesforce integration around a topology based on a consumer-producer, which is decoupled. In this topology Salesforce is both a producer and a consumer. Being a producer, it sends change events and platform messages in case of creating or altering records. As a consumer, it is sent inbound domain events by outside systems and reacts to them with automated workflow or record updates. ERP systems, data warehouses, and marketing services are external platforms that can be linked only by the backbone of the event. This model does not have any system-to-system dependencies. The backbone continues to receive events in a distributed and partitioned log that is used to ensure durable storage, ordered delivery, and complete consumer replay. The need behind this design is based on the log-based messaging principle, where the log is the sole source of truth for all intersystem communications. All participants read and write without involving other individuals. It is this independence that facilitates fault isolation and horizontal scalability in the integration environment [5].

Canonical event schema is a determinant of interoperability across systems connected. Notwithstanding the source, any event has a structured envelope. This envelope contains a special identifier, a timestamp, a domain classifier, an event type label, a version field, and a payload block. Another field that is

10.48047/jocaaa.2026.35.03.24

particularly significant is the version field. It enables the system to redirect the events to the appropriate schema handler in case the format changes with time. The schema registry implements the compatibility rules prior to any change being delivered to production consumers. The break changes are refused at the registry level. This eliminates base corruption of silent data to subsequent systems. The schema can be described as a stable integration contract, and all producers and consumers should respect this contract [6].

In this model, event sourcing and CQRS are adjusted to CRM domain structures. Aggregate roots are core Salesforce objects. All changes to the state cause a domain event, which is persisted to the backbone prior to any update being made to the records. This causes the event to be the authoritative history of all transactional history. The present record state is considered to be the derived projection of cumulative events. At the query side, CQRS divides the read and write tasks into different channels. Write-intent commands are sent through a command processor, which authenticates business rules and then executes any platform write. The read loads are provided based on materialized views that are supported by special consumer services. These perceptions are filled in in streams of domain events and are stored in exogenous query-optimized data stores. This isolation has continued to make reporting workloads compete with transactional operations using the same platform resources [5].

Idempotency processing is used to shield the pipeline against the impact of the same event being delivered two or more times. The common behavior in distributed streaming systems is at-least-once delivery. Duplicates are generated when the consumer is restarted, there is a network failure, or there is a broker failure. Every consumer in this model has a processing cache, the key of which is the canonical event identifier. The consumer will check this cache before any business logic is executed to find a previous record of processing. Duplicates that are confirmed are recognized and disregarded. The retention time of the cache is going to be longer than the maximum anticipated redelivery time. This is to ensure that no duplicated data is beyond the detection limit regardless of whether the system is operating normally or in degraded mode. The effect is effective—once processing behavior and no throughput cost of platform-level exactly-once guarantees [6].

The observability layer gives end-to-end visibility of the whole event topology. Each event envelope has a trace correlation identifier in it at the production point. This identifier is forwarded by all downstream processing hops without changes. It establishes a full causal chain, which also crosses system boundaries and domain boundaries. Instrumentation of every producer, broker, and consumer gathers timing information and transmits it to some centralized tracing backend. Monitoring of consumer lag is done at the level of partition. Automated scaling policies ensure that before service objectives are violated, lag is monitored and reacted upon by scaling before the targets are met. Periodically, the state of materialized views is reconciled with the state of live platform records. Inconsistencies within the system are identified and remedied prior to impacting downstream operations or deliverables to the customer [7].

The architecture has governance and security controls in its structural form. Only authorized service identities are given topic-level access permissions. Illegal manufacturers are unable to inject events, and the illegal consumers are unable to read sensitive streams. Encryption is done on payloads that carry personal or financial information, both at rest and in transit. The main management is assigned to a key store, which is controlled by the customers and can be rotated and revoked without re-encrypting past data. Any schema changes, as well as access control changes, are saved in an audit log, which is immutable. This log is kept in line with relevant regulatory schedules. The process of schema

10.48047/jocaaa.2026.35.03.24

compatibility validation is part of the development pipeline so that governance controls are only applied to the code before it gets to production as opposed to after a runtime violation has already taken place [7].

Schema Envelope Component	Architectural Function	Governing Mechanism
Unique Event Identifier	Enables idempotency detection and duplicate discarding at consumer level	Distributed in-memory processing cache
Schema Version Field	Routes events to correct schema handler as formats evolve over time	Schema registry compatibility enforcement
Domain Classifier and Event Type Label	Ensures correct topic routing and consumer subscription across heterogeneous systems	Role-based topic-level access control policies

Table 1: Canonical Event Schema Envelope Components and Their Architectural Functions. [7]

4: Comparative Analysis and Empirical Evaluation.

The empirical analysis was done in a global enterprise deployment of different business units on three continents. An average of ten million Salesforce record operations within sales, service, and field operations areas were processed by the environment every day. The scope of integration involved connections to an ERP platform, a cloud-based data warehouse, a marketing automation service, and a number of regional payment processing systems. An implementation process of a limited transition between the incumbent synchronous API-based integration architecture and the proposed event-driven one was carried out during a controlled transition time frame. There was also parallel operation in the migration process so as to allow direct side-by-side performance comparison at the same transactional loads. Point measurements were recorded during a prolonged pre-migration observation period, and after complete cutover, there were post-migration measurements recorded during an extended steady-state period. This configuration is indicative of known methods of testing the performance of distributed systems in full-scale production enterprise systems, in which controlled lab environments cannot match the complexity of actual integration workloads in multi-systems [8].

The performance of the system in the two architectural setups was tested in terms of four main metrics. Latency of integration was determined as the time that had passed between the production of events at the source system and the time the destination consumer confirmed that he received the event. System throughput was made as the amount of processed events that were successfully processed divided by a unit time during steady-state and peak load conditions. Fault tolerance was tested by inducing controlled failure injection of the event broker, individual consumer microservices, and the Salesforce API layer, and the mean time to recovery was documented in each of the failure conditions. The deployment agility was measured as the time of service failure during integration layer updates implemented on both architectural designs. These measures are all metrics that are most impactful on the performance of enterprise integration as well as consistent with evaluation frameworks developed in the distributed systems research to benchmark asynchronous messaging architectures [9].

Benchmark performance showed a significant and sustained performance improvement in the event-driven architecture in all the dimensions that were assessed. In the case of steady-state, the event-driven deployment realized an integration latency improvement of between three and five times that of the baseline synchronous architecture. Throughput measurements ensured that the event backbone supported

10.48047/jocaaa.2026.35.03.24

high event processing rates without compromising when the concurrency was raised, which could be explained by the partitioned consumer group model that allocates processing load among independently scalable consumer instances. Conversely, the synchronous architecture experienced throughput saturation beyond high levels of concurrency as API governor limit allocations were reached and a queue of requests built up. These results are in line with distributed messaging studies, which have shown that log-based streaming architecture can sustain a near-linear throughput scaling with increasing load, which request-response architectures intrinsically cannot structurally achieve beyond their concurrency limit [8].

The peak load analysis demonstrated the most operationally important difference between the two architectural configurations. The synchronous architecture was characterized by significant latency degradation and quantifiable rises in error rates of the transactions during high-volume processing periods that are associated with end-of-period business cycles. The connection of integration chains by casualty timers caused cascading failures to spread through the integration chain, increasing the effect of individual component slows in the system to the availability of a system-wide event. The event-based deployment ensured a consistent latency and almost zero error rates in similar peak operations. Auto-scaling of consumers addressed the lag in partitions that occurred in front of the violation of the service level objectives of latency, absorbing the increase in demand in the streaming layer without affecting the pressure on the Salesforce API limit. This failure containment pattern is an explicit result of the theoretical decoupling characteristics of asynchronous event propagation, where producer throughput is not directly connected to consumer processing capacity because of the durable event log [9].

The ability to upgrade to zero downtime was found to be valid on all integration updates that were implemented throughout the post-migration observation period. The blue-green pattern of consumer deployment allowed version transitions without offset regression or message loss because the version leaving processing purposefulness remained serving the active offset through to its present offset commitment until the new version became operational. Registry-level schema compatibility enforcement blocked incompatible schema change requests before they could reach production consumers and instead sent the requests through versioned migration paths, with no service interruption. Disaster recovery testing proved that recovery with full consumer state reconstruction based on event log replay was feasible within operationally acceptable recovery time constraints based on simulated data loss events. These results prove that, in comparison to synchronous architectures, asynchronous decoupling provides the disaster recovery capabilities that cannot be offered by durable event logs due to the availability of the destination system at the moment of the failure [10].

The empirical results are congruent with the theoretical predictions based on the suggested model of reference and are further extrapolated by the existing standards in the distributed system literature into the Salesforce SaaS-constricted framework. Studies of asynchronous messaging architecture have always suggested reduction of latency, fault isolation, and independent scalability as the main performance benefits of event-driven integration as compared to synchronous alternatives. The findings presented here attest to these benefits in a quantified fashion in a production enterprise setting in a context of governance limitations and API limits of implementation of a SaaS platform. The fact that results obtained in reality correspond to model forecasts makes it more probable that the reference model has a sound theoretical background and can be applied to enterprise deployments with similar integration profiles. It is also concluded that the findings provide a quantitative standard dataset of SaaS-specific EDA implementation currently unavailable in the published literature, which gives an empirical base that can be replicated in the future.

10.48047/jocaaa.2026.35.03.24

e comparative studies in this field [10].

Evaluation Metric	Synchronous Architecture Behavior	Event-Driven Architecture Behavior
Integration Latency	Pronounced degradation during peak load with cascading timeout failures across coupled chains	Stable latency maintained during peak periods through consumer auto-scaling on partition lag
System Throughput	Saturation at elevated concurrency as API governor limit allocations are exhausted	Near-linear scaling sustained under increasing load through partitioned consumer group distribution
Fault Tolerance	Individual component slowdowns propagate into system-wide availability events	Failure contained within affected subsystems without impact on adjacent integration domains

Table 2: Evaluation Metrics, Measurement Definition, and Architectural Configuration Comparison. [8]

Conclusion

This article has presented a comprehensive reference model for implementing event-driven architecture within large-scale Salesforce deployments. The model addresses a clearly defined gap in the existing literature on enterprise SaaS integration. Its core architectural contributions span six interdependent dimensions. A decoupled producer-consumer topology replaces brittle point-to-point synchronous dependencies with a resilient event backbone. A canonical event schema establishes a stable interoperability contract across heterogeneous systems. Event sourcing and CQRS are adapted to CRM domain structures, enabling immutable transactional history and independent read-write scaling. Idempotent processing delivers effective once semantics without throughput penalty. A distributed observability layer provides end-to-end traceability across all integration domains. Governance and security controls are embedded structurally rather than applied as operational afterthoughts. Together, these contributions form a coherent and implementable blueprint that organizations can adopt systematically rather than assembling piecemeal from disconnected architectural guidance.

The empirical findings reported in this paper confirm the model's operational effectiveness in a production enterprise environment. Integration latency was reduced by three to five times compared to the incumbent synchronous architecture. System throughput scaled sustainably under peak load conditions without the saturation behavior that exhausts API governor limits in conventional deployments. Fault tolerance improved measurably, with failures contained within affected subsystems rather than propagating across integration chains. Zero-downtime upgrades were consistently achieved, and disaster recovery was validated through event log replay under simulated data loss conditions. These results carry direct significance for enterprise Salesforce deployments operating under continuous availability requirements and escalating transaction volumes. They demonstrate that the performance advantages associated with event-driven integration in cloud-native environments are fully achievable within SaaS governance constraints when architectural adaptation is approached with the systematic rigor the proposed model provides.

For organizations modernizing legacy CRM environments, the implications of this work are practical and immediate. Many enterprises continue to operate synchronous Salesforce integrations not because of deliberate architectural preference but because accessible, validated guidance for event-driven alternatives

10.48047/jocaaa.2026.35.03.24

within SaaS constraints has not been available. This reference model removes that barrier. It provides a structured migration pathway that organizations can adopt incrementally, beginning with the highest-latency integration bottlenecks and expanding the event-driven topology progressively across the integration landscape. The model's governance and observability components also address the compliance and auditability requirements that frequently constrain architectural modernization decisions in regulated industries, making the transition viable in financial services, healthcare, and public sector contexts where data handling standards are most stringent.

Several limitations of the current study warrant acknowledgment. The empirical evaluation is drawn from a single enterprise deployment context within the consumer goods sector. Benchmark values will vary across organizations with different API usage patterns, event volume distributions, and infrastructure configurations. The model also presupposes access to enterprise-grade streaming infrastructure, which may require managed service substitution in organizations without dedicated platform engineering capacity. Future research should examine the model's performance characteristics across additional industry verticals, with particular attention to financial services and healthcare environments where transaction volumes and compliance requirements differ materially from the deployment studied here. Further investigation is warranted into the application of machine learning-based anomaly detection within the observability layer, enabling predictive failure identification that supplements the reactive alerting mechanisms described in this model.

The broader applicability of event-driven principles within enterprise SaaS ecosystems extends well beyond Salesforce. The canonical patterns formalized in this reference model—decoupled topology, schema-governed event contracts, idempotent consumers, and embedded observability—are transferable to any SaaS platform that exposes native publish-subscribe or change data capture capabilities. As enterprise software landscapes grow more complex and real-time engagement expectations continue to intensify, the structural limitations of synchronous integration will become operationally untenable across a widening range of platforms and industries. The formalization presented in this paper is intended to serve as a replicable foundation for that broader transition, equipping architects and engineers with the theoretical grounding and practical guidance necessary to design SaaS integration environments that are resilient, scalable, and genuinely capable of meeting the real-time demands of modern enterprise operations.

10.48047/jocaaa.2026.35.03.24

References

[1] Michael Affronti, "State of the Connected Customer," 6th ed., Salesforce, Inc., 2023. [Online]. Available: https://event.hbrturkiye.com/storage/uploads/state-of-the-connected-customer-655b114557dfa.pdf

[2] Hyeonsook Kim et al., "Towards Event-Driven Enterprise Architecture," ResearchGate, 2013. [Online]. Available: https://www.researchgate.net/publication/297364924_Towards_Event-Driven_Enterprise_Architecture

[3] Gwen Shapira et al., "Kafka: The Definitive Guide, 2nd Edition," O'Reilly Media, 2nd ed., 2021. [Online]. Available: https://www.oreilly.com/library/view/kafka-the-definitive/9781492043072/

[4] Sam Newman, "Building Microservices, 2nd Edition," O'Reilly Media, 2nd ed., 2021. [Online]. Available: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/

[5] Jay Kreps et al., "Kafka: a Distributed Messaging System for Log Processing," NetDB'11, Jun. 12, 2011. [Online]. Available: https://pages.cs.wisc.edu/~akella/CS744/F17/838-CloudPapers/Kafka.pdf

[6] Greg Young, "CQRS Documents." [Online]. Available: https://cqrs.wordpress.com/wp-content/uploads/2010/11/cqrs_documents.pdf

[7] Chris Richardson, "Microservices Patterns With Examples in Java," Manning Publications, 2018. [Online]. Available: https://www.manning.com/books/microservices-patterns

[8] Jay Kreps, "The Log: What every software engineer should know about real-time data's unifying abstraction," LinkedIn Engineering Blog, 2013. [Online]. Available: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

[9] Martin Kleppmann, "Designing Data-Intensive Applications," O'Reilly Media, 2017. [Online]. Available: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

[10] Tyler Akidau et al., "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," Proceedings of the VLDB Endowment, (2015). [Online]. Available: https://research.google/pubs/the-dataflow-model-a-practical-approach-to-balancing-correctness-latency-and-cost-in-massive-scale-unbounded-out-of-order-data-processing/