Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 3, Issue 1

Open Access | PP: 65-69

DOI: https://doi.org/10.70315/uloap.ulete.2026.0301011

Principles of Designing Event-Driven Systems for Real-Time Stream Data Processing

Sai Sruthi Puchakayala

Software Development Engineer III, Walmart, Centerton, Arkansas.

Abstract

The article examines design principles for event-driven systems that process continuous data streams under real-time constraints. Relevance follows from the shift of cloud applications toward low-latency, decoupled communication and from demand for consistent, fault-tolerant processing in distributed production environments. Novelty lies in synthesizing guidance from recent empirical and survey work on event-driven architectures, stream processors, checkpointing protocols, transactional stream processing, and cloud transaction models. The paper aims to systematize decisions on event modeling, time and ordering, state durability, delivery guarantees, scaling, and observability for production streaming pipelines. Methods combine comparative analysis of peer-reviewed studies, taxonomy construction, and pattern-based reasoning over representative architectures. Ten recent sources are reviewed, and a unified set of design recommendations and trade-offs is derived. The conclusion clarifies when to prioritize exactly-once processing, how to select checkpointing strategies, and how to align platform choices with business-critical workloads. The material supports engineers and architects building high-throughput streaming services.

Keywords: *Event-Driven Architecture, Stream Processing, Real-Time Data, Exactly-Once Semantics, Checkpointing, State Management, Transactional Stream Processing, Serverless, Microservices, Observability.*

INTRODUCTION

Real-time stream data processing has moved from a niche analytics capability to an operational backbone for cloud-native products, where business logic reacts to events rather than synchronous requests. In such systems, architectural choices directly determine correctness under failures, stability under skewed traffic, and the feasibility of scaling stateful computations without compromising latency or consistency guarantees. The purpose of this article is to formulate design principles for event-driven systems that ingest, transform, and emit streaming data with predictable correctness and operational control.

The study pursues three tasks:

1) to formalize design decisions around event representation, time/ordering, and state;
2) to analyze fault-tolerance strategies through the lens of checkpointing and transactional guarantees;
3) to derive engineering guidance that connects workload properties to platform and implementation choices.

Novelty is provided by an integrated synthesis of recent empirical evidence on the performance implications of event-driven decomposition, recent advances in state management for modern stream processors, and recent surveys that unify transactional and streaming guarantees within a single conceptual model.

MATERIALS AND METHODS

The evidence base is formed by ten recent works selected for direct relevance to event-driven stream processing and for complementary coverage of correctness, performance, and engineering practice: H. Cabane and K. Farias analyze performance impact of adopting event-driven architecture via an empirical comparison with a monolithic alternative [1]; M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos propose a transactional programming model for stateful serverless functions built on dataflow execution and compare against alternative backends [2]; M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos survey the evolution of stream processing systems across disorder, state, fault tolerance, and elasticity [3]; R. Laigner, G. Christodoulou, K. Psarakis, A. Katsifodimos, and Y. Zhou systematize building blocks and requirements for transactional cloud applications that frequently rely on event-driven messaging [4]; L. Lazzari and K. Farias formulate an agenda of research challenges for event-driven architecture grounded in a scoping review

Citation: Sai Sruthi Puchakayala, "Principles of Designing Event-Driven Systems for Real-Time Stream Data Processing", Universal Library of Engineering Technology, 2026; 3(1): 65-69. DOI: https://doi.org/10.70315/uloap.ulete.2026.0301011.

and case study [5]; Y. Mei and co-authors present Flink 2.0 disaggregated state management and quantify checkpointing and recovery improvements on benchmarks and production workloads [6]; G. Siachamis and co-authors experimentally evaluate checkpointing protocol families and report workload-dependent trade-offs between coordinated and uncoordinated approaches [7]; M. V. S. Silva and co-authors provide application guidelines for event-driven microservices under high data volumes via systematic mapping and multi-domain case evidence [8]; A. Yıldız and O. Demirörs introduce MicroArc as an analysis/design method to identify events and microservice candidates [9]; S. Zhang, J. Soto, and V. Markl synthesize transactional stream processing requirements, properties, and design options that merge ACID guarantees with streaming execution [10].

The article applies comparative analysis of the selected publications, constructs a consolidated taxonomy of design decisions (event model, time/ordering, state, delivery guarantees, recovery), and performs pattern-oriented synthesis to derive prescriptive principles and trade-offs, with particular attention to failure semantics and operational constraints documented in the sources.

RESULTS

Designing an event-driven system for real-time stream processing starts with a precise definition of an "event" as an engineering artifact rather than a loosely defined message. The transactional stream processing literature formalizes events as timestamped payloads that flow through continuous operators and windows, where state bridges the current and historical data and where schedulers and storage determine the attainable isolation and durability guarantees [10]. This framing is proper even outside strictly transactional workloads because it forces early decisions on what constitutes business state, what must be derived, and what must be persisted. In parallel, the broader survey of stream processing evolution emphasizes that modern systems must explicitly manage disorder (out-of-order arrivals), state growth, and recovery mechanisms as first-class design concerns rather than afterthoughts [3].

Event modeling must therefore encode both semantic intent and operational constraints. From a systems viewpoint, the event schema should separate immutable facts from derived projections, because recovery and replay rely on deterministic reconstruction of downstream state. The research agenda on event-driven architecture highlights practical difficulties that repeatedly arise when event streams become the primary integration mechanism: duplicates, out-of-sequence processing, collision-like effects when concurrent handlers update shared state, and monitoring complexity that grows with asynchronous fan-out [5]. Those issues do not originate solely from messaging; they stem from the interaction among event identity, handler idempotency, and the chosen processing guarantee. A robust principle is to treat idempotency as part of the domain contract: handlers must be safe under reprocessing within a bounded replay window, while the platform-level guarantee is selected to reduce, not eliminate, the need for defensive logic [7].

Delivery guarantees should be chosen by mapping business invariants to the observable failure modes. Exactly-once processing semantics are frequently pursued because they align failure recovery with failure-free execution at the operator-state level. Yet, the checkpointing study shows a critical nuance. There is a difference between exactly-once processing and exactly-once output observable by external sinks, which can still see duplicates after recovery unless sink-side deduplication or transactional sinks are used [7]. For business-critical operations that update shared mutable state (e.g., balances, reservations, inventory commitments), the transactional stream processing survey argues that streaming engines and databases in isolation struggle to satisfy both high-velocity ingest and ACID-style invariants; the combined model requires explicit choices about transaction scope, ordering guarantees, and state access patterns [10]. Hence, a practical principle is to separate pipelines into (i) analytical streams, where at least once plus idempotent aggregation often suffices, and (ii) operational streams, where transactional or exactly-once-aligned designs are justified, including explicit sink consistency mechanisms [7; 10].

Fault tolerance is not a single mechanism but a composition of checkpointing, replay, and state persistence. Empirical evidence on checkpointing protocols demonstrates that coordinated checkpointing (common in production-grade engines) tends to outperform uncoordinated and communication-induced approaches under uniformly distributed workloads. Yet, uncoordinated checkpointing can become competitive or even superior under skewed workloads, and it better accommodates specific cyclic query patterns that challenge coordinated designs [7]. This finding translates into a design principle: checkpointing strategy selection cannot be decoupled from workload shape (uniform vs. skewed), topology (acyclic vs. cyclic), and backpressure behavior. The system designer must therefore evaluate checkpoint overhead not only as periodic latency spikes, but as a coupling point between flow control, state size, and recovery time objectives [7].

State management is the dominant determinant of scalability for non-trivial streaming systems. Recent work on Apache Flink 2.0 proposes disaggregated state management that decouples computation from state storage by placing primary state in a remote distributed file system and using local storage as cache, combined with asynchronous execution. Reported evaluation results show substantial reductions in checkpoint duration and markedly faster recovery and rescaling relative to a baseline Flink 1.20 setup, indicating that state placement and state I/O scheduling can shift the cost structure of exactly-once pipelines [6]. As a principle, state architecture should be treated as a "budget allocation" problem: local state favors low-latency reads

but makes rescaling and recovery expensive; disaggregated state reduces reconfiguration and checkpoint costs but must mitigate remote I/O latency through caching and execution model adjustments [6].

Event-driven systems frequently operate as distributed applications rather than isolated pipelines, which brings transactional concerns into the core design. A tutorial on transactional cloud applications synthesizes three building blocks—programming model, runtime, and state management—tied to requirements such as fault tolerance and consistency. This lens is highly applicable to real-time retail and platform workloads where a single user action triggers multiple downstream effects (pricing, promotions, fraud screening, fulfillment) that must converge on consistent outcomes even when steps are asynchronous [4]. Furthermore, work on transactions across stateful serverless functions illustrates a concrete design path: using a dataflow system (via Flink StateFun) to centralize state management, messaging, and checkpointing, while providing a programming model that supports serializable distributed transactions with two-phase commit, as well as eventually consistent workflows via Sagas [2]. This supports a principle that is especially relevant to full-stack cloud engineering: when business logic spans multiple services, transactional boundaries must be explicit and matched to the runtime's recovery model; otherwise, the system accumulates hidden coupling through ad hoc compensations and fragile retry logic [2; 4; 10].

Figure 1 operationalizes these dependencies by showing how an event-driven streaming solution decomposes into building blocks and how requirements "attach" to them. The figure is a simplified adaptation of the building-block framing for transactional cloud applications [4], tailored to real-time stream data processing.

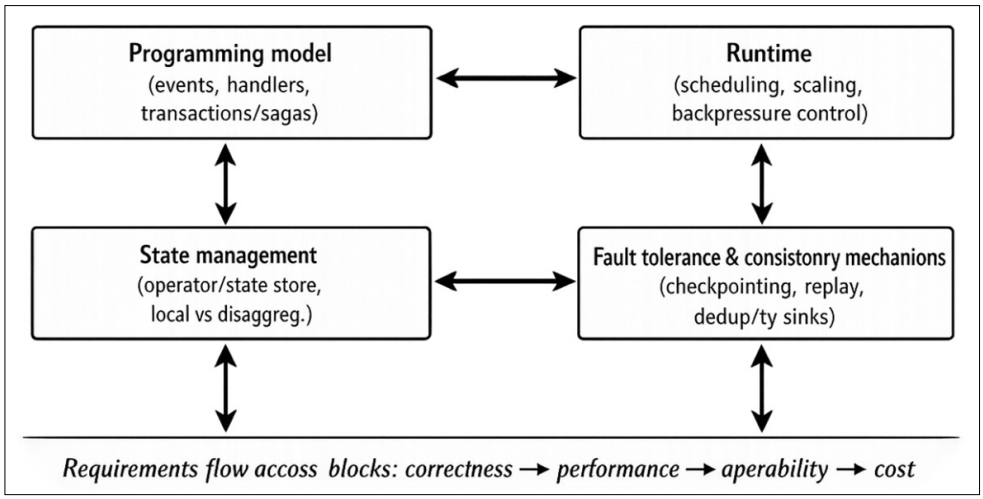

Figure 1. Building blocks and requirements for event-driven real-time stream processing (adapted from Laigner et al. [4])

Engineering practice benefits from explicit analysis/design methods that connect business processes to event flows and microservice boundaries. The MicroArc method formalizes early-phase modeling for identifying candidate events and microservices by representing process flows and extracting events as primary integration points [9]. When combined with guideline-oriented evidence from event-driven microservice studies emphasizing scalability and challenges in state consistency and testing under high volumes [8], a coherent principle emerges: event-driven stream processing succeeds when event boundaries align with stable business transitions and when state ownership is singular per invariant, while cross-invariant coordination is handled through transactions or compensating workflows rather than implicit shared updates [8–10].

Finally, performance expectations require disciplined interpretation. An empirical study comparing an event-based architecture against a monolithic implementation reports that, in the evaluated experimental setup, the monolithic alternative consumed fewer computational resources and achieved better response times, illustrating that decoupling via events can impose overheads that are not automatically offset by scalability benefits [1]. This evidence does not argue against event-driven design; it argues for treating event-driven decomposition as a controlled trade-off that must be justified by the need for independent scaling, fault isolation, or asynchronous workflow composition, and then engineered with attention to state, checkpointing, and operational tooling.

DISCUSSION

The results indicate that "event-driven" is not a single design choice but a set of coupled decisions whose success depends on workload characteristics, consistency requirements, and the dominant cost center (compute, state I/O, or operational complexity). For large-scale enterprise environments, a practical evaluation lens combines: the business-criticality of state updates (driving the need for transactional semantics), the expected traffic skew (shaping checkpointing choice), and the state size/reconfiguration frequency (shaping state placement) [6; 7; 10].

Table 1 summarizes the most consequential design choices and their expected system-level implications, grounded in the reviewed evidence.

Table 1. Design decision matrix for real-time event-driven streaming pipelines

Design decision	Option A	Option B	Engineering implication (evidence)
Processing guarantee	At-least-once	Exactly-once (operator-state)	Exactly-once reduces state anomalies but still requires sink-side strategy to achieve duplicate visibility after recovery [7].
Checkpointing protocol family	Coordinated checkpoints	Uncoordinated checkpoints	Coordinated tends to dominate on uniform workloads; uncoordinated can outperform on skewed workloads and may better support cyclic patterns [7].
State placement	Local embedded state	Disaggregated/remote primary state + cache	Disaggregation reduces checkpoint and reconfiguration cost but introduces remote I/O latency that must be mitigated by caching and execution model design [6].
Cross-service consistency	Sagas/ compensations	Serializable distributed transactions (2PC-style)	The transaction model must match failure recovery; stateful dataflows can provide transaction orchestration atop exactly-once execution, while Sagas target eventual consistency [2; 10].
Architectural decomposition	Event-driven microservices	Monolith/centralized services	Event-driven decomposition can increase overhead compared to a monolith in measured setups; justification should rest on scaling/fault isolation/workflow composition needs [1; 4].
Design methodology	Ad hoc event identification	Structured event-centric analysis/design	Formal methods can improve early identification of events and service candidates and reduce later refactoring driven by incorrect boundaries [9].

Correctness is purchased through stronger guarantees and more expensive coordination. At the same time, performance and cost are shaped by state architecture and by how much coordination is pushed into the runtime versus the application. In production settings, the failure model must be explicitly stated in requirements because it determines whether the system requires transactional stream processing properties or can remain within conventional streaming semantics [10].

Table 2 links recurring engineering concerns to practices and to the strongest supporting sources among the selected literature.

Table 2. Engineering concerns, recommended practices, and supporting evidence

Concern in real-time streaming	Practice aligned with the evidence base	Primary supporting sources
Workload skew and backpressure are affecting recovery	Benchmark checkpointing families under skew; avoid assuming coordinated checkpoints are always optimal	[7]
Large state with frequent rescaling	Prefer disaggregated state designs with cache + async execution; treat state I/O as a first-order performance dimension	[6]
Multi-service invariants (payments, reservations, inventory commitments)	Use transactional stream processing framing; define transaction scope and ordering constraints explicitly	[10; 4]
Stateful “serverless” workflows	Centralize state, messaging, and checkpointing in a dataflow runtime; select 2PC vs Sagas based on isolation needs	[2]
Event boundary identification	Apply a structured method to derive events from business process flows and to identify service candidates	[9]
Performance expectations during decomposition	Treat event-driven decomposition as a hypothesis; measure overhead and justify by scalability or fault isolation objectives	[1]
Organizational/operational adoption risks	Plan for monitoring complexity and “hidden coupling” introduced by asynchronous fan-out; prioritize observability design early	[5]
High-volume microservice communication	Use guideline-based patterns for implementing event-driven microservices and explicitly manage state consistency/testing.	[8]

The implications for an enterprise-focused software development profile (Java/Spring microservices, distributed systems, cloud platforms) follow naturally: the highest leverage comes from enforcing explicit contracts for event identity and idempotency, selecting checkpointing and state placement with realistic traffic models, and treating transactional boundaries as part of the architecture rather than as application-level patches. Recent literature reinforces that many failures in event-driven systems are not “broker problems” but specification problems: unclear state ownership, ambiguous ordering expectations, and insufficiently engineered recovery paths.

CONCLUSION

The study's purpose was realized through a coherent set of design principles that connect event modeling, time/ordering, state architecture, delivery guarantees, and operational control into one engineering decision space. Formalizing events addressed the first task by timestamped payloads processed by continuous operators, whose correctness depends on explicit state ownership and clearly specified ordering assumptions. The second task was addressed by grounding fault tolerance in the choice of checkpointing protocol and by linking transactional requirements to runtime-level support for recovery and consistency. The third task was addressed by translating recent empirical findings into actionable guidance: event-driven decomposition requires justification and measurement; disaggregated state can materially improve checkpointing and recovery when engineered with latency mitigation; checkpointing strategy selection depends on workload shape; transactional stream processing concepts clarify when operational invariants require stronger guarantees than conventional streaming pipelines provide.

REFERENCES

1. Cabane, H., Farias, K., & Kleinner. (2024). On the impact of event-driven architecture on performance: An exploratory study. *Future Generation Computer Systems, 153*, 52–69. https://doi.org/10.1016/j.future.2023.10.021
2. de Heus, M., Psarakis, K., Fragkoulis, M., & Katsifodimos, A. (2022). Transactions across serverless functions leveraging stateful dataflows. *Information Systems, 108*, 102015. https://doi.org/10.1016/j.is.2022.102015
3. Fragkoulis, M., Carbone, P., Kalavri, V., et al. (2024). A survey on the evolution of stream processing systems. *The VLDB Journal, 33*, 507–541. https://doi.org/10.1007/s00778-023-00819-8
4. Laigner, R., Christodoulou, G., Psarakis, K., Katsifodimos, A., & Zhou, Y. (2025). Transactional cloud applications: Status quo, challenges, and opportunities. In *Companion of the 2025 International Conference on Management of Data (SIGMOD/PODS '25)* (pp. 829–836). Association for Computing Machinery. https://doi.org/10.1145/3722212.3725635
5. Lazzari, L., & Farias, K. (2023). *Uncovering the hidden potential of event-driven architecture: A research agenda* (arXiv:2308.05270). arXiv. https://arxiv.org/abs/2308.05270
6. Mei, Y., Lan, Z., Huang, L., Lei, Y., Yin, H., Xia, R., Hu, K., Carbone, P., Kalavri, V., & Wang, F. (2025). Disaggregated state management in Apache Flink 2.0. *Proceedings of the VLDB Endowment, 18*(12), 4846–4859. https://doi.org/10.14778/3750601.3750609
7. Siachamis, G., Psarakis, K., Fragkoulis, M., Van Deursen, A., Carbone, P., & Katsifodimos, A. (2024). Checkmate: Evaluating checkpointing protocols for streaming dataflows. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)* (pp. 4030–4043). IEEE.
8. Silva, M., Santos, L. F. C., Soares, M., & Rocha, F. G. (2025). Guidelines for the application of event-driven architecture in microservices with high volume of data. In *Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025)* (Vol. 2, pp. 859–866). SciTePress. https://doi.org/10.5220/0013348600003929
9. Yıldız, A., & Demirörs, O. (2025). MicroArc: Event driven analysis and design method for microservices. *Procedia Computer Science, 263*, 583–590. https://doi.org/10.1016/j.procs.2025.07.070
10. Zhang, S., Soto, J., & Markl, V. (2024). A survey on transactional stream processing. *The VLDB Journal, 33*, 451–479. https://doi.org/10.1007/s00778-023-00814-z

Copyright: © 2026 The Author(s). This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.