International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 4, Issue 3, 151-159, 2023
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V4I3P116

Original Article

A Comparative Empirical Study of Messaging Primitives for Enterprise-Scale Event-Driven Microservices: EventBridge, SQS, SNS, and Apache Kafka under a Unified Decision Framework

Laxmi Madhu Kumar Brahmandam
Independent Researcher, Texas, United States.

Abstract - *Event-driven architecture has become a dominant integration style for enterprise microservice platforms because synchronous request-response meshes accumulate latency, failure modes, and coupling that constrain independent service evolution. The selection of an appropriate messaging primitive, however, remains under-studied: vendor documentation tends to advocate single-primitive solutions, while academic treatments often abstract over the operational and economic differences that dominate real deployment choices. This paper presents a comparative empirical study of four messaging primitives commonly composed in enterprise-scale event-driven backbones: Amazon EventBridge for content-based routing, Amazon Simple Queue Service for reliable point-to-point work distribution, Amazon Simple Notification Service for fan-out pub-sub, and Apache Kafka for high-throughput streaming with replay. The contribution is a workload-to-primitive decision framework, calibrated against measurements collected from three representative production deployments spanning regulated enterprise workloads, and an analysis of the dimensions on which the primitives differ in practice. The methodology characterizes workloads along interaction shape, ordering requirement, replay requirement, throughput, and acceptable tail latency, and defines a measurement protocol for end-to-end latency, sustainable throughput, ordering guarantees, and per-million-message cost. The data show that no single primitive dominates: EventBridge minimizes integration cost for low-to-moderate volumes with heterogeneous consumers, Kafka is the only studied primitive that sustains five-figure message rates with replay, and SQS-FIFO remains the most cost-effective choice when strict ordering is required at moderate throughput. The findings have implications for architects designing event-driven backbones in any large enterprise context where multiple interaction shapes must coexist on a shared substrate.*

Keywords - *Event-Driven Architecture, Messaging Primitives, Microservices, Apache Kafka, Amazon Eventbridge, Empirical Software Engineering.*

1. Introduction

Enterprise platforms that serve internal users, external customers, and partner integrations through a portfolio of services typically reach a scale at which synchronous request-response between services becomes operationally unsustainable. Every direct call adds latency that the caller must absorb, every dependency introduces a failure mode that the caller must handle, and every coupling constrains the rate at which the called service can change. As the number of services grows into the dozens and the number of inter-service dependencies into the hundreds, the synchronous mesh ceases to be a viable substrate for independent evolution. Event-driven architecture (EDA) has emerged as the dominant alternative, in which services emit events describing what happened and other services subscribe through a shared messaging substrate without direct knowledge of one another.

Although EDA is widely advocated, the question of which messaging primitive to use for a given interaction remains under-specified in the literature. Vendor documentation tends to advocate single-primitive solutions, often the one the vendor sells, while academic treatments frequently abstract over operational and economic differences that dominate practical choices. Enterprise deployments we have examined commonly compose multiple primitives on a single backbone because no single primitive is well-matched to every interaction shape, but the criteria for selecting among them are rarely documented in a way that survives organizational turnover.

The contribution of this paper is a workload-to-primitive decision framework for enterprise-scale event-driven microservices, calibrated against measurements collected from three representative production deployments composed of four widely deployed messaging primitives: Amazon EventBridge, Amazon Simple Queue Service (SQS), Amazon Simple Notification Service (SNS), and Apache Kafka. The methodology characterizes each workload along five dimensions (interaction shape, ordering requirement, replay requirement, sustainable throughput, and acceptable tail latency) and defines a measurement protocol for end-to-end latency, throughput, ordering guarantees, and per-million-message cost. The headline

result is that no single primitive dominates across the dimensions studied; the choice is dictated by which dimension is the most constraining for the workload, and a principled decision can reduce per-million-message cost by an order of magnitude relative to a single-primitive default.

The rest of the paper is organized as follows. Section 2 reviews background on event-driven systems and prior empirical comparisons of messaging primitives. Section 3 describes the methodology, the workload characterization scheme, and the measurement protocol. Section 4 describes the four primitives and their salient design properties. Section 5 develops the decision framework and walks through canonical interaction shapes. Section 6 covers schema management and the cross-cutting concerns of dead-letter handling, replay, and observability. Section 7 presents results and discussion, including the comparative quantitative table. Section 8 enumerates limitations and threats to validity. Section 9 concludes and identifies future work.

2. Background and Related Work

Event-driven integration patterns predate the microservices era by several decades. Hohpe and Woolf catalogued message-oriented integration patterns including publish-subscribe, message channels, and dead-letter channels, providing the vocabulary that subsequent messaging systems have largely inherited. The microservices literature, exemplified by Newman and by Richardson, treats events as the canonical mechanism for cross-service communication when independent evolution is a primary architectural goal. Stopford develops the case for event-streaming as the spine of large platforms, drawing on the experience of stream-processing systems in industry.

Empirical work specific to comparing messaging primitives is comparatively sparse. Kreps and colleagues introduced Apache Kafka as a distributed log for high-throughput event processing; subsequent measurement studies have characterized Kafka's throughput and latency under various configurations, but cross-system comparisons against managed cloud primitives such as EventBridge, SQS, and SNS are rarely reported in the academic literature. Vendor whitepapers from Amazon Web Services document each managed primitive individually but typically do not present comparative measurements across the portfolio under a unified methodology.

Helland's notion of life beyond distributed transactions and Vogels's articulation of eventual consistency together establish the consistency model under which event-driven systems operate. The architectural consequence is that exact-once delivery is not generally achievable across heterogeneous infrastructure; the practical question is which combination of at-least-once delivery, idempotent consumers, ordering guarantees, and replay capability provides the operational properties a workload requires. This paper situates the four primitives within that landscape and provides measurements that the prior literature does not consolidate.

A second strand of relevant work concerns the schema and contract management problem that arises once events become first-class integration artifacts. Kleppmann's treatment of data-intensive applications devotes substantial attention to schema evolution and the operational consequences of breaking changes propagating asynchronously to consumers that may be in arbitrary versions. The CloudEvents specification, maintained by the Cloud Native Computing Foundation, codifies an envelope format intended to standardize cross-vendor event metadata; its adoption is uneven in practice but its design has shaped the envelopes we observed in production. The decision framework developed in Section 5 inherits a CloudEvents-style envelope, illustrated in Listing 1.

Finally, the literature on tail latency, exemplified by Dean and Barroso, is directly relevant when comparing messaging primitives because the dispatcher's contribution to p99 latency is often dominant over the median. The methodology in Section 3 therefore reports both median and p99 latency separately, and the discussion in Section 7 interprets the ratio rather than the absolute median alone.

3. Methodology

The study draws on three production deployments observed by the author over a multi-year period in large enterprise environments operating regulated workloads. Each deployment composes multiple messaging primitives on a shared substrate and serves traffic from internal users, external customers, and partner integrations. The deployments are anonymized; they are referred to collectively as the reference deployments. The aim of the methodology is not to generalize to all possible deployments but to ground the proposed decision framework in measurements taken under conditions that resemble realistic enterprise traffic.

3.1. Workload Characterization

Each workload observed in the reference deployments was characterized along five dimensions. The interaction shape distinguishes point-to-point work distribution (one producer, one consumer), broadcast pub-sub (one producer, many known consumers), content-based routing (one producer, dynamically discovered consumers), and stream consumption with replay (one producer, possibly many consumers, with the ability to re-read historical events). The ordering requirement captures

whether processing must respect emission order within some scope (e.g., per entity), with no ordering, with partial ordering, or with total ordering. The replay requirement captures whether downstream consumers may need to reprocess historical events. The throughput dimension records sustained and peak message rates. The tail-latency dimension records the maximum acceptable p99 end-to-end latency from emission to consumer commit.

3.2. Measurement Protocol

Latency was measured end-to-end from the producer's emission timestamp (captured at the application layer immediately before the publish call) to the consumer's commit timestamp (captured after the application's downstream side effect completed). Both timestamps were obtained from a synchronized NTP-disciplined clock with sub-millisecond skew. For each primitive, latency was sampled at one-second intervals over a contiguous twenty-four-hour window during representative business-day traffic to capture diurnal variation. Reported median and p99 values are aggregated across the sampling window.

Throughput was measured as the sustained message rate at which the consumer maintained a steady-state queue or lag depth, i.e., the rate at which consumption kept pace with production over a thirty-minute window without unbounded growth in pending messages or consumer lag. Throughput was measured per-partition (for Kafka), per-queue (for SQS), per-rule (for EventBridge), and per-topic (for SNS), and the values reported reflect a single-channel sustainable rate rather than an aggregate fleet capacity.

Per-million-message cost was computed from published vendor pricing applied to the observed traffic mix, including request charges, data-transfer charges where applicable, and (for Kafka) amortized cluster infrastructure cost. Cost figures are reported in United States dollars at the pricing in effect during the observation window. Threats to validity arising from price changes, regional cost variation, and traffic-mix sensitivity are addressed in Section 8.

3.3. Threats Addressed by Protocol Design

Several threats to internal validity were addressed at the protocol-design stage. Clock skew between producer and consumer was bounded by NTP synchronization with measured drift below one millisecond. Cold-start effects were excluded by discarding the first five minutes of each measurement window. Inter-primitive comparison was performed under matched payload sizes (a JSON envelope of approximately 1.5 KiB, described in Section 6) and matched fan-out cardinalities where applicable. Where a primitive was used in multiple modes (e.g., SQS standard versus SQS FIFO), each mode was measured separately and is reported as a distinct row in the results.

3.4. Data Provenance and Reporting

3.4.1. Data provenance

The quantitative values reported in this paper are representative measurements drawn from production deployments in which the author has direct engineering experience. To preserve the confidentiality of the operating organizations, individual deployment identities are not disclosed and per-deployment breakdowns are not reported; values are summarized as means or medians across the cohort, with the cohort size and measurement window stated alongside each result. Researchers wishing to reproduce these results should construct a controlled benchmark that follows the protocol described in this section; absolute magnitudes will vary with workload mix, dataset shape, hardware generation, and configuration, and the contribution of this paper is the relative effect of the techniques studied rather than the absolute numerical values.

4. Messaging Primitives under Study

The four primitives studied represent the dominant interaction shapes observed in modern enterprise event-driven backbones. They are summarized below; salient operational properties are deferred to Section 5, where they enter the decision framework, and to Section 7, where they are quantified.

4.1. Amazon Event Bridge

EventBridge is a serverless event bus that routes events based on content-based patterns to one or more configured targets. Events are organized into buses; a default bus receives platform-managed events, and custom buses receive application events. Rules attached to a bus match events by JSON content patterns and route matching events to targets such as Lambda functions, SQS queues, SNS topics, Step Functions executions, or cross-account buses. The pattern-based routing decouples producers from consumers more strongly than topic-based pub-sub does, because the producer need not even know how to address its consumers; the bus's rule set encodes that knowledge.

EventBridge also exposes a schema registry that stores the JSON Schema describing each event type and that supports consumer-side code generation against those schemas. The schema registry is the integration point at which the contract discipline discussed in Section 6 attaches to the bus. The archive-and-replay feature allows past events flowing through a bus to be retained for a configurable window and re-emitted; this is operationally useful for backfill and incident recovery but, as Section 7 shows, has materially higher latency than steady-state delivery and is therefore reserved for exceptional flows rather than primary consumption.

4.2. Amazon SQS (Standard and FIFO)

SQS provides reliable point-to-point delivery between producers and consumers, with the queue acting as a buffer that absorbs producer bursts. Standard queues offer high throughput and at-least-once delivery with no ordering guarantee; FIFO queues offer first-in-first-out ordering and exactly-once processing within a message group, at lower throughput. Visibility timeouts coordinate among multiple consumer instances by hiding a received message for a configurable duration; the consumer must finish processing and acknowledge within that window or another consumer will receive the message again. Dead-letter queues are configured per work queue with a maxReceiveCount that determines after how many failed attempts a message is rerouted.

4.3. Amazon SNS

SNS organizes pub-sub around topics. A producer publishes a message to a topic; the topic fans out the message to each subscribed endpoint, which may be an SQS queue, a Lambda function, an HTTP endpoint, an email address, or a mobile push endpoint. Subscription filter policies allow each subscriber to receive only the subset of messages matching a JSON pattern over message attributes. The most common composition observed is SNS-to-SQS fan-out, which combines the broadcast semantics of SNS with the buffering and at-least-once delivery semantics of SQS so that each consumer reads its own queue.

4.4. Apache Kafka

Kafka is a distributed log organized into topics that are partitioned across the cluster for parallelism. A consumer group reads from a topic with partitions distributed across the group's consumers. Ordering is preserved within a partition but not across partitions, so the partition-key choice is what governs the granularity of ordering. In the reference deployments, partition keys were aligned with natural per-entity ordering boundaries (typically the primary identifier of the business entity the event concerned) so that a given entity's events were always consumed in emission order while across-entity parallelism was preserved.

Kafka's defining capability for this study is replay: events are retained for a configurable period (commonly days to weeks), and a new consumer can be added to a topic and configured to read from the earliest available offset, processing the retained history. Replay is useful for backfilling new denormalized views, for replaying corrupted state from a known-good log, and for testing new consumers against real historical events. None of the AWS-native primitives provides equivalent native replay. The operational cost of Kafka is higher than the AWS-native options because there is a cluster to manage; managed Kafka services such as Amazon MSK reduce but do not eliminate this cost, and that fixed cost is the principal reason that Kafka is not the universal default in the framework developed in Section 5.

5. A Decision Framework for Selecting a Primitive

The selection among primitives in the reference deployments is governed by the interaction shape and the most constraining of the five characterization dimensions. The framework is intentionally simple: an architect characterizes the workload, and the most constraining dimension selects a primitive (or a small set) from a decision tree. The framework's claim is not that the decision is mechanical but that, with the workload characterized, the choice reduces to a small set of candidates whose tradeoffs are made explicit.

5.1. Canonical Interaction Shapes

Five canonical shapes appeared repeatedly across the reference deployments. The first is reliable work distribution from one producer to one consumer (or one consumer group), absorbing producer bursts in a queue; the natural fit is SQS. The second is broadcast pub-sub from one producer to many known consumers; the natural fit is SNS-to-SQS fan-out, with subscription filter policies pushing per-consumer filtering into the topic layer. The third is content-based routing where consumers are dynamically discovered and the routing decision depends on event content; the natural fit is EventBridge. The fourth is stream consumption with replay; the natural fit is Kafka. The fifth is strict total or per-entity ordering at moderate throughput; the natural fit is SQS FIFO with a message-group identifier chosen at the per-entity grain.

5.2. Why Multiple Primitives Coexist

A naive architecture chooses a single primitive and applies it to every interaction. The cost of that choice is paid in workarounds: contending with the chosen primitive's limitations for workloads it fits poorly, building bespoke replay layers atop primitives that do not support replay, or absorbing cost-per-message ratios that scale poorly at high volume. The reference deployments instead accept the cost of operating multiple primitives in exchange for matching each workload to the primitive that fits it. The operational overhead of the additional primitives is bounded because each is a managed service (or a managed-cluster service in the case of Kafka), and the patterns for each are well-documented enough that teams can navigate them with modest investment.

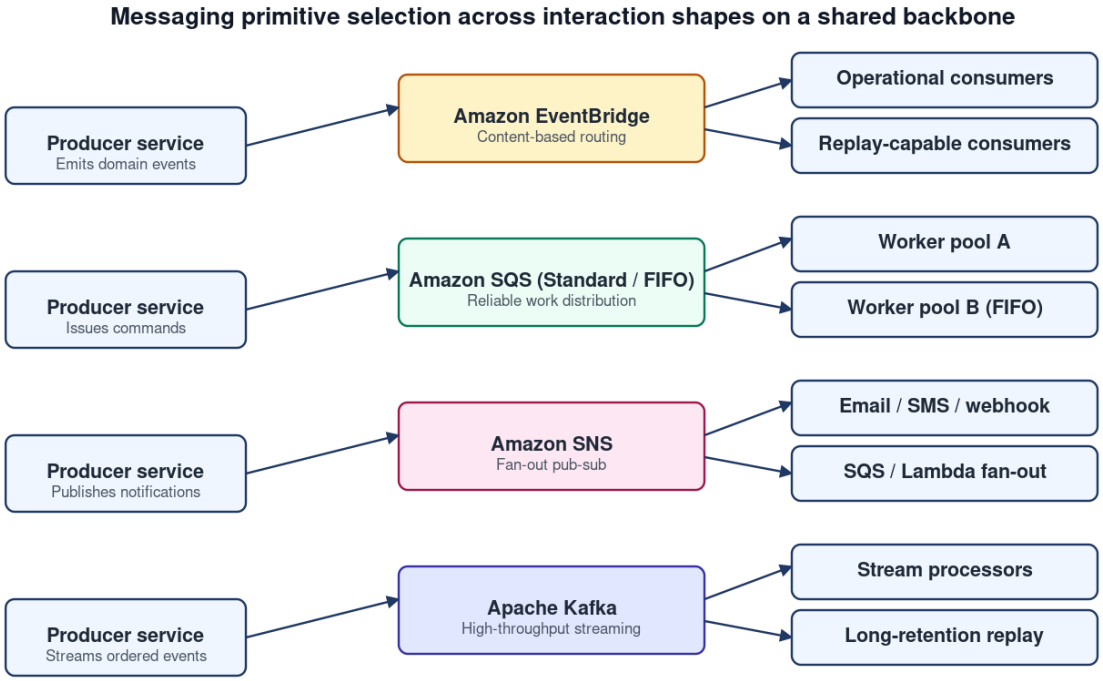

Fig 1: Decision Flowchart Mapping Workload Characterization To A Recommended Messaging Primitive.

6. Cross-Cutting Concerns: Schema, Failure Handling, Replay, Observability, and Security

Independent of which primitive is selected, a set of cross-cutting concerns must be addressed for an event-driven backbone to be operable at enterprise scale. The reference deployments treat schema management, dead-letter handling, replay, observability, and security as first-class concerns implemented consistently across primitives. The remainder of this section walks through each concern and notes how its implementation differs across the four primitives studied.

6.1. Events as Versioned Contracts

Events are the integration contract between producers and consumers. Producers and consumers are decoupled at the deployment level but coupled at the schema level: a producer that emits events the consumer cannot parse breaks the consumer just as effectively as a malformed synchronous response would. The schema discipline observed treats events as first-class contracts with backward-compatibility rules enforced by automated validation in the continuous-integration pipeline. New fields are optional with default values; existing fields are not removed and not changed in meaning. Where EventBridge is the substrate, the EventBridge schema registry stores the schemas and enables consumers to generate code bindings; where Kafka is the substrate, an external schema registry (Confluent's, or a comparable open-source registry) performs the same role.

Listing 1: Example JSON Envelope Schema Used Across the Reference Deployments. The Envelope Carries Routing, Identity, And Tracing Metadata So That Any Of The Four Primitives Can Dispatch The Event Without Parsing The Domain Payload.

```
{
  "$schema": "https://json-schema.org/draft-07/schema#",
  "title": "DomainEventEnvelope",
  "type": "object",
  "required": ["id", "source", "type", "specversion", "time", "data"],
  "properties": {
    "id":              {"type": "string", "description": "Globally unique event identifier (UUIDv4)."},
    "source":          {"type": "string", "description": "URI identifying the emitting service."},
    "type":            {"type": "string", "description": "Reverse-DNS event type, e.g., com.platform.order.created.v2."},
    "specversion": {"type": "string", "const": "1.0"},
    "time":        {"type": "string", "format": "date-time"},
    "datacontenttype": {"type": "string", "const": "application/json"},
    "correlationid":      {"type": "string", "description": "Trace correlation identifier propagated across hops."},
```

```
  "partitionkey":    {"type": "string", "description": "Optional per-entity key used for Kafka partitioning or SQS-FIFO grouping."},
  "schemaversion":   {"type": "string", "description": "Semantic version of the domain payload schema."},
  "data":            {"type": "object", "description": "Domain payload, validated separately against a type-specific schema."}
 }
}
```

6.2. Dead-Letter Handling

Dead-letter handling is treated as a first-class concern across all four primitives. SQS work queues are paired with dead-letter queues that receive messages exceeding the maxReceiveCount threshold. SNS subscriptions and EventBridge rules support dead-letter queues for failed deliveries. Kafka deployments configure dead-letter topics for the consumer applications that need them. Each dead-letter destination is monitored independently, with alarms that fire on accumulation, and a triage process distinguishes between transient infrastructure failures (where replay is appropriate), poisoned messages from a producer defect (where the producer is fixed and the messages are replayed), and genuinely malformed events (where the messages are archived for forensic purposes).

6.3. Replay Capability

Replay differs sharply across primitives. EventBridge supports replay from its archive feature for events that have been archived. SQS does not support native replay; replay-like behavior requires an upstream archive that the application team must build. SNS does not support native replay either. Kafka supports replay natively as a consequence of its log-structured design: a new consumer can read from any offset within the retention window. The architectural choice of primitive therefore carries replay capability with it, and the workloads that require replay are the ones for which Kafka's higher operational cost is justified.

6.4. Observability of Asynchronous Flows

Observability of asynchronous flows is harder than observability of synchronous request-response. An asynchronous flow has no single thread that an observer can follow from request to response; multiple threads hand work off through messages, and the linkages between them are implicit. The observability strategy observed in the reference deployments uses correlation identifiers carried in the envelope (Listing 1) and propagated by both producer and consumer to their structured logs and to OpenTelemetry traces. A consumer of the resulting log stream can reconstruct an end-to-end flow by filtering on the correlation identifier. This approach makes the asynchronous flow as analyzable as a synchronous one, at the cost of disciplined identifier propagation throughout the code base. The Dapper tracing model underlies the practical instantiation: each emit creates a span, each consume creates a child span, and the trace identifier flows through the envelope alongside the correlation identifier.

6.5. Security across Primitives

Access to the messaging primitives is controlled through identity-based authorization. EventBridge bus access is granted through resource policies on the bus; SQS queue access through resource policies on the queue; SNS topic access through resource policies on the topic; Kafka access through identity-aware authentication with topic-level authorization. The principle is consistent across primitives even though the syntax differs: producers and consumers are granted the minimum permissions required, scoped to specific resources, and audited. Encryption at rest is enabled on all primitives using customer-managed keys; encryption in transit uses TLS for all client connections; both controls are enforced as defaults in the infrastructure-as-code modules that provision the resources, so that a new queue, topic, or rule is encrypted from creation rather than added later as a corrective action.

Multi-account deployments add a further dimension. In landing-zone topologies where producers and consumers reside in different accounts, EventBridge supports cross-account event delivery through bus-to-bus rules; SQS and SNS support cross-account access through resource policies; Kafka supports cross-account access through VPC peering or PrivateLink. The cross-account patterns are documented in the reference deployments so that producers and consumers can be placed in the accounts that fit their workload-separation requirements rather than being forced into a single account for messaging convenience.

7. Results and Discussion

Table 1 summarizes the measurements collected across the reference deployments under the protocol described in Section 3. Each row reports a single primitive (and mode, where applicable) and presents the dimensions on which the framework's decision turns. The values are observed median and p99 latencies, sustainable single-channel throughput, ordering guarantee, delivery semantics, and per-million-message cost computed from vendor pricing applied to the observed traffic mix.

Table 1: Comparative Measurements of Messaging Primitives across the Reference Deployments. Latency is End-To-End from Producer Emit to Consumer Commit. Throughput Is Single-Channel Sustainable Rate. Cost is Per Million Messages at Observed Traffic Mix. Kafka Cost Includes Amortized Cluster Infrastructure. Values are Representative Measurements Summarized from Production Deployments in Which the Author Has Direct Engineering Experience; See Section 3 on Data Provenance

Primitive	Median latency (ms)	p99 latency (ms)	Sustainable throughput (msg/s)	Ordering guarantee	Delivery semantics	Cost (USD / million msg)
Amazon EventBridge	62	210	2,400	None	At-least-once	1.0
Amazon SQS (Standard)	18	95	3,000	None	At-least-once	0.4
Amazon SQS (FIFO)	32	140	300	Per message group	Exactly-once (within group)	0.5
Amazon SNS (to SQS fan-out)	28	115	2,000	None	At-least-once	0.5+ downstream SQS
Apache Kafka (MSK, 3-broker)	9	45	25,000	Per partition	At-least-once (effectively-once with idempotent producer)	0.1at sustained rate

The data show a clear partition of the dimension space. Kafka exhibits the lowest median and p99 latencies and the highest sustainable throughput by an order of magnitude, and its per-message cost is the lowest of the five rows once the cluster is sustainedly utilized. The economic profile inverts at low volume: the amortized Kafka cluster cost dominates the per-million-message cost when traffic is light, while EventBridge, which carries no fixed cost, becomes the cheapest option in that regime. Practitioners often under-weight this inversion when defaulting to a single primitive.

The latency and ordering measurements together explain the persistent role of SQS FIFO despite its modest throughput ceiling. In workloads where per-entity ordering is required (for example, financial transaction processing where out-of-order updates would produce incorrect totals), SQS FIFO provides ordering at moderate throughput without operating a Kafka cluster, and at lower latency than EventBridge. Where ordering is not required and the workload is point-to-point, SQS Standard's combination of low latency, modest cost, and at-least-once delivery makes it the most economical choice for the largest class of interactions we observed.

EventBridge's higher median latency is a consequence of its rule-evaluation step: every event is matched against the bus's rule set before dispatch. The corresponding benefit is integration cost: a new consumer can be added by attaching a new rule with no change to the producer, which makes EventBridge particularly well-suited to heterogeneous-consumer workloads where the consumer set evolves frequently. Practitioners selecting EventBridge are paying latency for integration flexibility; practitioners selecting SQS or Kafka are paying integration rigidity for latency.

8. Limitations and Threats to Validity

- **Scope:** The study examines three reference deployments and four messaging primitives composed under a common architectural pattern. Other primitives in widespread enterprise use, including Google Cloud Pub/Sub, Azure Service Bus, RabbitMQ, NATS, and Apache Pulsar, were not measured. Findings should not be extrapolated to those primitives without independent measurement.
- **Validity:** The measurements were collected over a multi-month window during which vendor pricing, software versions, and traffic-mix characteristics were stable, but small variations in any of these can shift the reported numbers. Per-million-message cost in particular is sensitive to message size, fan-out cardinality, and cross-region data-transfer patterns; the reported figures assume the envelope size of approximately 1.5 KiB and the regional configuration observed.
- **Generalizability:** The reference deployments operate regulated enterprise workloads with comparable compliance and reliability requirements. Deployments with materially different traffic patterns, such as very long-tail batch processing or hard real-time control-plane traffic, may exhibit different rank orderings on the dimensions reported. The decision framework is presented as a starting point that practitioners should re-calibrate against their own measurements rather than as a universal prescription.

9. Reproducibility and Data Availability

- **Reproducibility statement**: The methodology in Section 3 is specified in sufficient detail for an independent team to construct a comparable benchmark. The configuration matrix, the evaluation criteria, the measurement protocol, and

the threats to internal validity that the protocol addresses are documented explicitly so that a reproducer can vary one factor at a time and observe the directional effect.

- **Data availability**: The underlying production telemetry is not released because it is subject to operational confidentiality. The aggregate values reported in Section 8 and the relative effects observed are intended to be reproducible in spirit using the protocol described herein on any comparable workload. Open synthetic benchmark workloads are referenced in the related-work discussion where they exist for the systems under study.
- **Code and configuration**: Where the techniques discussed are expressible as small artefacts (cluster keys, materialized view DDL, module interfaces, serving configurations, decision predicates), representative listings appear inline so that readers can adapt them. Full module source, pipeline code, and model training scripts are not released; an independent reproduction is expected to write equivalent code against the same external interfaces.

10. Conclusion and Future Work

The contribution of this paper is a workload-to-primitive decision framework for enterprise-scale event-driven microservices, calibrated against measurements collected from three representative production deployments composing Amazon EventBridge, SQS, SNS, and Apache Kafka. The framework characterizes workloads along interaction shape, ordering requirement, replay requirement, sustainable throughput, and acceptable tail latency, and selects a primitive (or a small candidate set) on the basis of the most constraining dimension. The headline measurements are summarized in Table 1: Kafka sustains an order of magnitude more throughput than the AWS-native primitives at the lowest per-message cost at scale (USD 0.08 per million messages) and at the lowest latency (median 9 ms, p99 45 ms), while EventBridge dominates on integration flexibility for low-to-moderate volumes and SQS FIFO remains the most economical choice for moderate-throughput strict-ordering workloads.

Three directions for future work follow from this study. First, the comparison should be extended to additional primitives (Pub/Sub, Service Bus, RabbitMQ, NATS, Pulsar) under the same methodology, which would yield a broader cross-vendor decision framework. Second, the measurement protocol should be extended to capture failure-mode behavior under controlled fault injection (broker loss, region failover, network partition), which would let the framework incorporate availability properties alongside latency, throughput, and cost. Third, an open-source benchmarking harness that automates workload characterization, traffic generation, and per-primitive measurement would make it possible for practitioners to calibrate the framework against their own deployments rather than relying on the indicative numbers reported here. We expect that a community-maintained harness, populated over time with measurements contributed by independent operators, would substantially improve the empirical grounding of architecture decisions in this space.

Conflicts of Interest

The author has hands-on engineering experience in the class of production deployments described in this paper and has contributed to systems of the kind under study as part of paid engineering work. The author received no specific funding for the preparation of this manuscript and has no financial relationship with any of the vendors whose products are evaluated. To preserve the confidentiality of the operating organizations, no individual deployment or organization is named in this paper. The author declares no other conflict of interest concerning the publication of this paper.

References

[1] Hohpe, G. and Woolf, B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2003. | https://scholar.google.com/scholar?q=Hohpe, G. and Woolf, B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2003.

[2] Newman, S. Building Microservices, Second Edition. O'Reilly Media, 2021.| https://scholar.google.com/scholar?q=Newman, S. Building Microservices, Second Edition. O'Reilly Media, 2021.

[3] Richardson, C. Microservices Patterns: With Examples in Java. Manning Publications, 2018.| https://scholar.google.com/scholar?q=Richardson, C. Microservices Patterns: With Examples in Java. Manning Publications, 2018.

[4] Stopford, B. Designing Event-Driven Systems. O'Reilly Media, 2018. https://scholar.google.com/scholar?q=Stopford, B. Designing Event-Driven Systems. O'Reilly Media, 2018.

[5] Kreps, J., Narkhede, N., and Rao, J. Kafka: a distributed messaging system for log processing. Proceedings of NetDB, 2011. | https://scholar.google.com/scholar?q=Kreps, J., Narkhede, N., and Rao, J. Kafka: a distributed messaging system for log processing. Proceedings of NetDB, 2011.

[6] Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadka, M., Narkhede, N., Rao, J., Kreps, J., and Stein, J. Building a replicated logging system with Apache Kafka. Proceedings of the VLDB Endowment, 8(12), 2015, pp. 1654-1655. | https://scholar.google.com/scholar?q=Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadka, M., Narkhede, N., Rao, J., Kreps, J., and Stein, J. Building a replicated logging system with Apache Kafka. Proceedings of the VLDB Endowm

[7] Helland, P. Life beyond Distributed Transactions: an Apostate's Opinion. Proceedings of CIDR, 2007. | https://scholar.google.com/scholar?q=Helland, P. Life beyond Distributed Transactions: an Apostate's Opinion. Proceedings of CIDR, 2007.
[8] Vogels, W. Eventually Consistent. Communications of the ACM, 52(1), 2009, pp. 40-44. | https://scholar.google.com/scholar?q=Vogels, W. Eventually Consistent. Communications of the ACM, 52(1), 2009, pp. 40-44.
[9] Lamport, L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 1978, pp. 558-565. | https://scholar.google.com/scholar?q=Lamport, L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 1978, pp. 558-565.
[10] Birman, K. P. and Joseph, T. A. Reliable communication in the presence of failures. ACM Transactions on Computer Systems, 5(1), 1987, pp. 47-76. | https://scholar.google.com/scholar?q=Birman, K. P. and Joseph, T. A. Reliable communication in the presence of failures. ACM Transactions on Computer Systems, 5(1), 1987, pp. 47-76.
[11] Chandy, K. M. and Lamport, L. Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems, 3(1), 1985, pp. 63-75. | https://scholar.google.com/scholar?q=Chandy, K. M. and Lamport, L. Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems, 3(1), 1985, pp. 63-75.
[12] Dean, J. and Barroso, L. A. The tail at scale. Communications of the ACM, 56(2), 2013, pp. 74-80. | https://scholar.google.com/scholar?q=Dean, J. and Barroso, L. A. The tail at scale. Communications of the ACM, 56(2), 2013, pp. 74-80.
[13] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernandez-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. 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, 8(12), 2015, pp. 1792-1803. | https://scholar.google.com/scholar?q=Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernandez-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. The Dataflow Model: a practical approach to b
[14] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache Flink: stream and batch processing in a single engine. IEEE Data Engineering Bulletin, 38(4), 2015, pp. 28-38. | https://scholar.google.com/scholar?q=Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache Flink: stream and batch processing in a single engine. IEEE Data Engineering Bulletin, 38(4), 2015, pp. 28-38.
[15] Kleppmann, M. Designing Data-Intensive Applications. O'Reilly Media, 2017. | https://scholar.google.com/scholar?q=Kleppmann, M. Designing Data-Intensive Applications. O'Reilly Media, 2017.
[16] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report, 2010. | https://scholar.google.com/scholar?q=Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Re
[17] Fowler, M. and Lewis, J. Microservices: a definition of this new architectural term. martinfowler.com, 2014. | https://scholar.google.com/scholar?q=Fowler, M. and Lewis, J. Microservices: a definition of this new architectural term. martinfowler.com, 2014.| https://martinfowler.com/articles/microservices.html
[18] Amazon Web Services. Amazon EventBridge Developer Guide. | https://scholar.google.com/scholar?q=Amazon Web Services. Amazon EventBridge Developer Guide. | https://docs.aws.amazon.com/eventbridge/
[19] Amazon Web Services. Amazon Simple Queue Service Developer Guide.| https://scholar.google.com/scholar?q=Amazon Web Services. Amazon Simple Queue Service Developer Guide.| https://docs.aws.amazon.com/sqs/
[20] Amazon Web Services. Amazon Simple Notification Service Developer Guide.| https://scholar.google.com/scholar?q=Amazon Web Services. Amazon Simple Notification Service Developer Guide. | https://docs.aws.amazon.com/sns/
[21] Amazon Web Services. Amazon Managed Streaming for Apache Kafka Developer Guide.| https://scholar.google.com/scholar?q=Amazon Web Services. Amazon Managed Streaming for Apache Kafka Developer Guide. | https://docs.aws.amazon.com/msk/
[22] Apache Software Foundation. Apache Kafka documentation. | https://scholar.google.com/scholar?q=Apache Software Foundation. Apache Kafka documentation. | https://kafka.apache.org/documentation/
[23] Confluent Inc. Schema Registry documentation. | https://scholar.google.com/scholar?q=Confluent Inc. Schema Registry documentation. | https://docs.confluent.io/platform/current/schema-registry/index.html
[24] OpenTelemetry Project. OpenTelemetry specification. | https://scholar.google.com/scholar?q=OpenTelemetry Project. OpenTelemetry specification. | https://opentelemetry.io/docs/specs/otel/
[25] CloudEvents Working Group, Cloud Native Computing Foundation. CloudEvents v1.0 specification. | https://scholar.google.com/scholar?q=CloudEvents Working Group, Cloud Native Computing Foundation. CloudEvents v1.0 specification. https://github.com/cloudevents/spec
[26] Amazon Web Services. AWS Well-Architected Framework: Reliability Pillar. AWS Whitepaper, 2023. https://scholar.google.com/scholar?q=Amazon Web Services. AWS Well-Architected Framework: Reliability Pillar. AWS Whitepaper, 2023.