SvedbergOpen
DISSEMINATION OF KNOWLEDGE
International Journal of Artificial Intelligence and Machine Learning
Publisher's Home Page: https://www.svedbergopen.com/
Research Paper Open Access

Architecting Event-Driven Banking Systems For Real-Time Financial Transaction Processing

Jyothirmai Gurramula

Independent Researcher, USA. ORCID ID: https://orcid.org/0009-0004-1600-0897

Abstract
Real-time payment processing in enterprise banking demands message delivery guarantees that go beyond what default Kafka configurations offer. A poorly configured Kafka producer can silently duplicate a debit instruction; a misconfigured consumer can drop a settlement event without logging an error. Addressing both failure modes at once, without introducing unacceptable throughput penalties, requires treating reliability as a coordinated system design problem rather than a per-component configuration checklist. This paper proposes the Four-Pattern Framework (4PF), a deployment-sequenced architecture combining four interoperable Kafka and Spring Boot patterns into a coherent reliability stack for financial transaction workloads. The first pattern hardens the producer layer using enable.idempotence=true, a scoped transactional.id, acks=all, and single-in-flight request serialization to prevent broker-side duplication. The second pattern embeds a deduplication fingerprint check at the application tier, protecting against the redelivery failures that producer-side configuration cannot intercept. The third pattern replaces Kafka's timer-based offset progression with AckMode.MANUAL_IMMEDIATE, ensuring that committed offsets track confirmed processing outcomes rather than elapsed time. The fourth pattern separates command and query responsibilities through CQRS event sourcing, maintaining an immutable payment log that satisfies PSD2 Article 32's five-year retention mandate. Production data from fintech deployments handling cross-bank B2B reconciliation and real-time interbank clearing validate the framework's reliability claims. Settlement latency dropped from over 24 hours to sub-five-second processing windows, with zero duplicate payment events observed across the monitored deployment period. The 4PF gives practitioners a concrete, sequenced path from basic streaming to auditable exactly-once financial processing.

Keywords: Apache Kafka; event-driven architecture; financial transaction processing; exactly-once semantics; event sourcing; CQRS; Spring Boot; Four-Pattern Framework.

1. Introduction

Duplicate charges and dropped settlement instructions are not abstract reliability concerns in financial systems. They produce real balance errors, failed clearing cycles, and regulatory findings. A bank's payment processing layer typically channels a single transaction through fraud scoring, liquidity validation, ledger write, clearing network submission, and settlement confirmation, each handled by a separate service. Any one of those hops can receive the same event twice if the message broker does not enforce delivery guarantees at the protocol level.

Kafka's exactly-once delivery mechanisms exist, but activating them is not a matter of flipping a single flag. Three producer-level settings must be enabled together; each one in isolation produces a weaker guarantee than their combination, and that gap is where most production systems fail (Raptis & Passarella, 2023). The difference between a correctly configured and a partially configured Kafka producer is invisible during normal operation — it only surfaces during network retries and consumer restarts, precisely the moments when a payment system cannot afford surprises.

The shift from batch processing to Kafka-based event streaming collapsed settlement windows from 24 to 48 hours down to seconds in the deployments Kennady and Mayilsamy (2022) studied. That latency improvement is real and measurable. What is less visible is the reliability gap the shift created: Kafka's default at-least-once delivery gives consumers no protection against receiving a message multiple times after a rebalance or restart.

The financial consequences of that gap are serious. A duplicated debit is a fraud liability. A dropped clearing event is a settlement failure. Neither shows up as an exception in the application log when Kafka's default commit behavior is in effect.

The gap does not stem from any single misconfiguration. It stems from the fact that Kafka's exactly-once guarantees require three producer-level settings to be active simultaneously, plus a separate application-layer deduplication mechanism, plus manual offset commit discipline, plus a read/write model separation to protect the audit record. These four requirements interact. Enabling idempotent production without manual offset commit does not produce exactly-once semantics. Manual offset commit without application-layer deduplication does not survive consumer crash scenarios. Any subset of the four leaves an exploitable failure path open.

The Four-Pattern Framework (4PF) proposed here closes all four failure paths in a deployment sequence validated against production evidence. Pattern 1 configures the producer for exactly-once writes. Pattern 2 handles the application-layer deduplication that producer configuration cannot cover. Pattern 3 implements manual offset management with dead-letter routing. Pattern 4 applies CQRS event sourcing to protect the audit trail. The sequencing is not arbitrary: it reflects the operational frequency of each failure mode, so institutions can deploy each pattern incrementally and see measurable reliability gains before moving to the next.

Two production implementations ground the framework's empirical claims. A fintech B2B reconciliation platform running high-volume cross-bank Kafka workloads, documented by Chirukuri (2025), supplied the throughput and latency benchmarks cited throughout this paper. Spring Boot microservice patterns for the producer and consumer layers came from Deshpande (2025). The remaining cited sources contribute theoretical background and comparative evidence. Section 2 covers the literature across all four pattern domains; Section 3 specifies the 4PF; Section 4 walks through the Spring Boot implementation; Section 5 covers operational reliability and compliance; Sections 6 and 7 discuss limitations and conclude.

2. Literature Review

2.1 Reliability Failures Specific to Financial Event-Driven Systems

Batch-processing systems dominated financial transaction infrastructure for decades because they offered one thing streaming systems historically did not: a natural transaction boundary. Every payment in a batch either committed as a unit or rolled back as a unit. Dayarathna and Perera (2018) tracked the progressive replacement of batch architectures with stream-based event processing across enterprise systems, noting that the latency reduction was substantial but came with softer default consistency guarantees. Kleppmann (2017) gave that trade-off a formal vocabulary: at-least-once delivery means duplicates are possible; at-most-once means drops are possible; exactly-once means neither, but only under specific protocol conditions that must be explicitly configured.

Kennady and Mayilsamy (2022) measured the migration from batch to near-real-time payment processing at two financial institutions and reported settlement latency reductions of over 90%. What their work also revealed was the operational complexity introduced when consumers restart mid-processing: without manual offset management, a crashed consumer could leave gap transactions neither acknowledged nor reprocessed. Taibi, Lenarduzzi, and Pahl (2018) studied architectural pattern combinations for microservices and concluded that event sourcing paired with CQRS is architecturally necessary for systems requiring both query performance and audit continuity, not merely a design preference. That finding directly motivates Pattern 4 of the 4PF.

2.2 Exactly-Once Semantics: What Kafka Actually Guarantees

Kafka's exactly-once semantics arrived in version 0.11. The mechanism pairs two cooperating components. The idempotent producer receives a producer ID from the broker at startup and attaches a monotonically increasing sequence number to every message per partition. Brokers detect and discard produce requests that share a previously seen PID-sequence pair, eliminating the duplicates that TCP-level retries would otherwise create. The transactional producer extends that guarantee across partition boundaries by enrolling writes in a named transaction context; a single atomic commit either succeeds across all target partitions or rolls back entirely.

Raptis and Passarella (2023) surveyed production Kafka deployments and found that genuine exactly-once guarantees required all three of enable.idempotence=true, a unique transactional.id per producer instance, and max.in.flight.requests.per.connection=1 to be active together. Enabling idempotence without a transactional ID

produces only per-partition deduplication. Enabling a transactional ID with high in-flight request counts introduces message reordering that breaks the sequence number monotonicity the protocol relies on. Each partial configuration creates a silent reliability gap: the system behaves correctly in most conditions and fails precisely during the network errors and restart scenarios it was configured to handle.

Whether transactional Kafka configuration is worth its throughput overhead depends on the workload. Pallaprolu (2025) worked through the pattern selection criteria for modern distributed systems and drew a line between use cases where EOS justifies its cost and those where at-least-once delivery is acceptable. Payment initiation and settlement clearing fall clearly on the EOS side of that line; notification and audit fan-out topics do not. Pattern 1 uses that distinction to scope its configuration: EOS settings apply only to financial topics, leaving non-critical topics at lower overhead.

2.3 Consumer Restarts and the Application-Layer Deduplication Gap

Producer-side EOS does not reach into the consumer's failure modes. Consider a consumer that reads a payment message, writes the debit record to the downstream database, and then crashes before committing its Kafka offset. When the consumer restarts, Kafka redelivers the uncommitted message. The producer delivered it exactly once. The consumer processed it twice. Without application-layer deduplication, the database receives two writes for one transaction.

Spring Boot Kafka integrations are particularly exposed to this failure because Spring's default listener configuration does not prevent double-processing on restart. Venkatachalapathi (2024) traced the problem to offset commit timing and showed that binding offset advancement to explicit application acknowledgment — using AckMode.MANUAL_IMMEDIATE — is the configuration change that closes the gap between Kafka delivery guarantees and application-level processing correctness.

An empirical survey of production microservice architectures by Laigner et al. (2024) put numbers on how often this gap is left open: consumer-side idempotency was missing more frequently than producer-side EOS across the systems they examined. Practitioners apparently treat producer configuration as the primary reliability lever and treat consumer protection as secondary. The 4PF inverts that ordering — Pattern 2 deploys before Pattern 1 — because consumer restarts are simply more frequent than producer-side network failures in production financial environments.

A production-grade EDA system integrating Kafka with Camunda BPM, reported by Odofin et al. (2024), showed how programmatic transaction management can coordinate dead-letter routing with downstream compensating events through event headers alone, without requiring a distributed transaction coordinator. That architectural approach shaped Pattern 3's DLQ classification design.

2.4 Event Sourcing, CQRS, and Financial Audit Requirements

Overeem, Spoor, and Jansen (2017) studied the operational costs of event sourcing in production systems and identified schema evolution as the primary long-term risk: when event schemas change after a long-running topic has accumulated records, consumer applications must deserialize records written under multiple historical schema versions simultaneously. That finding shapes the 4PF's schema registry requirement in Pattern 4. Every payment event must be enrolled in a schema registry before log compaction is activated. Retroactive schema changes after compaction can corrupt historical records that compaction has already collapsed.

Distributed transaction coordination in banking APIs presents a choice between saga-based compensation and two-phase commit. Hebbar (2025) put both approaches under load in high-volume banking scenarios and found that saga compensation recovered faster and at lower throughput cost. That result is what justifies Pattern 3's DLQ-triggered compensation design over a more heavyweight rollback mechanism.

Event-driven microservices built on event sourcing principles keep each service's state isolated from shared database schemas, as Bellemare (2020) showed — a structural property that matters for payment systems where schema changes in one service should not break audit consumers in another. Applying those principles to financial transaction systems specifically, Narayanan (2025) measured reductions in inter-service coupling and elimination of the synchronous call chains responsible for latency spikes under load. Across the broader EDA pattern landscape, Vallabhaneni (2025) identified CQRS combined with event sourcing as the ceiling configuration for distributed systems where regulatory audit continuity is a hard requirement — not one architectural option among many, but the only design that satisfies both query and compliance constraints simultaneously.

3. Four-Pattern Framework (4PF) for Financial Transaction Reliability

The 4PF structures Kafka reliability engineering into four discrete but interdependent patterns. Each one closes a specific failure mode. Deploying all four in sequence produces financial-grade exactly-once processing with a complete, tamper-evident audit trail. Table 1 lays out the full pattern matrix.

Table 1. Four-Pattern Framework: Failure Modes, Configuration Targets, and Deployment Order

Pattern	Name	Failure Addressed	Key Configuration	Deploy Order
P1	Exactly-Once Producer	Broker-level duplicate writes on network retry	enable.idempotence=true; transactional.id; acks=all; max.in.flight=1	First
P2	Idempotent Consumer	Double-write on consumer crash before offset commit	App-layer fingerprint check; dedup store (Redis or DB)	Before P1 in high-restart environments
P3	Offset-Managed Recovery	Silent message loss from timer-based auto-commit	enable.auto.commit=false; AckMode.MANUAL_IMMEDIATE; DLQ routing	After P2 stabilized
P4	CQRS Event Sourcing	Audit record corruption; command/query model coupling	Read/write topic separation; schema registry; log compaction; isolation.level=read_committed	Last — after P1-P3 are stable

3.1 Pattern 1: Exactly-Once Producer

Getting the producer configuration right means activating four settings together. None of the four is optional if exactly-once delivery is the target. The settings are:

```
enable.idempotence=true
transactional.id=payment-svc-${instanceId}
acks=all
max.in.flight.requests.per.connection=1
```

The idempotence flag activates per-partition sequence tracking at the broker. Each produce request carries a PID and sequence number; brokers reject any request that duplicates a previously acknowledged PID-sequence pair. The transactional.id enrolls the producer in Kafka's transaction coordinator, enabling atomic multi-partition commits and activating the epoch-fence mechanism that terminates zombie producers after coordinator failovers. Setting acks=all blocks produce acknowledgment until all in-sync replicas have written the record; combined with max.in.flight=1, this prevents the message reordering that higher in-flight counts would introduce against the idempotent sequence numbering.

Every one of these settings carries a throughput cost. Synchronous ISR acknowledgment adds write latency proportional to the slowest replica. Single-in-flight serialization reduces per-partition peak throughput. The practical mitigation is scope: Pattern 1 applies only to payment initiation, settlement, and clearing topics. Non-financial topics carrying notification or configuration events stay at acks=1 with idempotence disabled. Scoped application of EOS keeps the throughput impact manageable without compromising the guarantees where they matter.

3.2 Pattern 2: Idempotent Consumer

Pattern 1 cannot protect against what happens after delivery. A consumer that processes a payment record and writes it to the database, then crashes before committing its Kafka offset, will receive that record again on restart. Kafka did not duplicate it. The consumer's crash recovery created the duplicate. This is a distinct failure class requiring a distinct solution.

The solution is a fingerprint store. Before writing any payment record downstream, the consumer computes a deduplication key from the transaction ID and the partition-offset pair, then checks whether that key already exists in a Redis cache or a relational deduplication table. A cache hit means the record was already processed; the consumer acknowledges the message to Kafka and discards it without a downstream write. A miss triggers

normal processing: compute, fingerprint store write, downstream database write, offset commit — in that order.

Laigner et al. (2024) found application-layer idempotency absent in the majority of production EDA systems they examined. That absence was more common than incomplete producer configuration, suggesting that practitioners deprioritize consumer-layer protection when they have achieved producer-side EOS. The 4PF's sequencing recommendation inverts that priority: because consumer restarts are operationally far more frequent than producer network failures, Pattern 2 should be deployed and validated before Pattern 1's transactional overhead is accepted.

3.3 Pattern 3: Offset-Managed Failure Recovery

Kafka's auto-commit timer fires every five seconds by default. When it fires, it advances the committed offset for all messages that have been fetched since the last commit, regardless of whether those messages have been successfully processed. A consumer that fetches fifty records, processes thirty, and then crashes will have its offset advanced by the timer to cover all fifty if the timer fires during that window. When it restarts, it picks up at offset fifty-one. Records thirty-one through fifty were never processed. They are gone. No error was logged.

Disabling auto-commit and switching to AckMode.MANUAL_IMMEDIATE is what Pattern 3 does. Under that mode, Acknowledgment.acknowledge() is called by the application only after the downstream database write has confirmed success. The Kafka client does not move the offset pointer until that call is made. A crash at any point before acknowledgment leaves the offset uncommitted; the consumer restarts and re-receives the message, which Pattern 2's fingerprint check then handles without a double write. The max.poll.records=50 ceiling keeps the re-processing window bounded at fifty records per crash event.

Messages that exhaust their retry budget are routed to a dead-letter queue topic rather than blocking the partition. A dedicated DLQ consumer classifies each routed record: transient failures (network timeouts, temporary unavailability) are re-queued after exponential backoff; persistent failures (bad schema, unresolvable account ID) trigger a compensating payment-failed event on the command topic, notifying downstream services that the transaction cannot complete. Hebbar (2025) showed that Saga-based compensation of this kind outperforms two-phase commit rollback in banking API workflows on both throughput and recovery speed.

3.4 Pattern 4: CQRS Event Sourcing

The audit failure mode Pattern 4 addresses is subtler than duplication or data loss. It is corruption through coupling. When consumers write aggregated state back to the same topic they read from, or when the operational query model shares schema with the compliance record, a schema migration on the query side can invalidate the historical audit log. CQRS separates those concerns by design.

Every payment state change generates an immutable event on the payment-events topic: initiated, fraud-scored, authorized, clearing-submitted, settled, or failed. The command side only writes; it never reads its own topic. The query side consumes that log and builds read-optimized materialized views in a separate store. The event log itself is never modified. PSD2 Article 32 mandates five-year retention with tamper evidence; Kafka's append-only, offset-immutable structure satisfies the tamper-evidence requirement, and a log.retention.ms setting aligned to five years satisfies the duration requirement.

Schema evolution must be managed proactively. Before enabling log compaction on the payment-events topic, all event types must be enrolled in a schema registry. Each appended event carries its schema ID as a header. Consumers can deserialize records written under any historical schema version by resolving the ID against the registry. Overeem, Spoor, and Jansen (2017) identified schema evolution management as the primary operational liability in long-lived event-sourced systems; the schema registry requirement in Pattern 4 directly addresses their finding.

4. Implementation with Spring Boot and Spring Kafka

4.1 Producer Configuration

Spring Kafka's ProducerFactory accepts the four Pattern 1 properties as standard Kafka client configuration keys. Two implementation details that documentation does not prominently surface are worth noting. First, the transactional.id must be scoped to the service instance identifier, not just to the service name. Horizontal scaling with a shared transactional.id causes transaction epoch conflicts between replicas: when two instances

share an ID, one will fence the other as a zombie producer during a coordinator rebalance. Using a deployment-specific suffix (service name + pod ID or hostname) prevents that conflict.

Second, transactional writes must be wrapped in a KafkaTransactionManager-controlled transaction boundary. Spring's @Transactional annotation propagates the transaction context across both the Kafka producer and any JPA or JDBC database writes within the same method boundary. A database write failure causes both the database change and the in-flight Kafka transaction to be rolled back together. That joint rollback is what CQRS event sourcing requires at the command side: the payment record in the database and the payment event in Kafka must either both commit or both abort.

4.2 Consumer Configuration

ConcurrentKafkaListenerContainerFactory requires two explicit settings that are easily missed. The first is enable.auto.commit=false in the Kafka consumer properties. Spring Kafka 2.x sets this internally in some configurations, but it does not propagate the setting to the Kafka client properties in all containerized deployment environments.

Setting enable.auto.commit=false directly in the properties map passed to the ConsumerFactory removes any ambiguity about whether Spring's internal setting is taking effect. The second required property is AckMode.MANUAL_IMMEDIATE on the container factory. This mode commits the current message's offset immediately after Acknowledgment.acknowledge() is called, without waiting for the end of a batch or a poll cycle. The combination of explicit auto-commit disable and manual-immediate acknowledgment gives the application full control over offset progression. Table 2 lists the complete production parameter set used in both reference deployments.

Table 2. Kafka Configuration Parameters: 4PF Production Values and Financial Rationale

Parameter	Value	Pattern	Rationale
enable.idempotence	true	P1	Activates PID/sequence dedup at broker; prerequisite for EOS
transactional.id	payment-svc-{instanceId}	P1	Instance-scoped; prevents epoch conflicts between scaled replicas
acks	all	P1	ISR-wide acknowledgment; no data loss on leader failover
max.in.flight.requests.per.connection	1	P1	Prevents reordering that breaks idempotent sequence numbering
enable.auto.commit	false	P3	Disables timer-based offset advance; required for manual control
AckMode	MANUAL_IMMEDIATE	P3	Binds offset commit to confirmed downstream write completion
max.poll.records	50	P3	Caps reprocessing window per consumer crash event
isolation.level	read_committed	P4	Filters uncommitted transactional records from consumer read path

4.3 Infrastructure and Cluster Operations

Both reference deployments ran on GCP with Helm-managed Kafka clusters provisioned through GCP Cloud Builder pipelines. Three broker-level settings are non-negotiable for financial workloads.

A replication factor of 3 paired with min.insync.replicas=2 keeps the acks=all guarantee from becoming a production bottleneck: two replicas in-sync is sufficient for a produce request to complete, so a single slow or temporarily offline replica does not stall the write path. On the retention side, log.retention.ms at 157,680,000,000 with log.cleanup.policy=compact+delete gives each payment key a preserved latest state while the configured ceiling enforces storage bounds. Partition count must match the maximum expected consumer parallelism before the topic sees production traffic; partition additions post-launch require topic migration work that can disrupt ordering guarantees for in-flight records.

Consumer pod count was governed by horizontal pod autoscaling against the consumer_lag JMX metric. The autoscaler configuration from Mondal et al. (2023) for Kafka consumer workloads set the scaling floor at two

replicas and the ceiling at the partition count, so the consumer group never exceeded its parallelism ceiling. Cross-availability-zone broker replication followed the multi-cloud streaming topology described by George (2022), keeping payment event processing uninterrupted during single-AZ failures.

5. Operational Reliability and Compliance

5.1 Consumer Lag Monitoring

The gap between the offset of the last produced message and the offset the consumer has committed is what Kafka calls consumer lag. Under normal load, that gap stays small — new messages arrive, get processed, and get committed quickly enough that lag never accumulates. When lag grows outside of scheduled maintenance windows, something has gone wrong: a consumer is processing too slowly, a consumer process has crashed without recovering, or a burst of incoming messages has temporarily outpaced the consumer group's processing capacity. Those three causes look similar in lag charts but require different responses, which is why lag must be correlated with consumer group member count and CPU metrics before acting.

Monitoring for 4PF deployments collects three signal families: Kafka JMX metrics covering consumer lag per partition, producer record-send-rate, and broker request latency; Spring Boot Actuator metrics covering listener container thread utilization and DLQ routing rate; and application-layer metrics covering deduplication cache hit rate and downstream write latency percentiles. Table 3 maps each signal to its operational function and compliance retention requirement.

Table 3. Observability Matrix: Signal Collection, Operational Function, and Retention Requirements

Signal	Source	Operational Use	Compliance Use	Retention
Consumer lag / partition	Kafka JMX	Detects slow/crashed consumers; triggers pod scaling	Processing continuity evidence	90 days
DLQ routing rate	Spring Actuator	Identifies error-class trends; triggers ops alerts	Failed transaction audit trail	5 years
Dedup cache hit rate	Micrometer gauge	Quantifies redelivery frequency; validates P2	Idempotency mechanism activity log	1 year
Producer txn abort rate	Kafka JMX	Detects transactional producer instability	EOS configuration compliance evidence	1 year
Payment event retention	log.retention.ms + compaction	Storage management	PSD2 Art. 32: 5-year immutable record	5 years (mandatory)

5.2 Dead-Letter Queue Event Classification

Across the two reference deployments, DLQ events sorted empirically into three classes. Transient failures made up approximately 73% of DLQ volume: network timeouts, temporary downstream service unavailability, and Kafka consumer group rebalances that interrupted mid-batch processing. These were successfully reprocessed after a configurable delay without operator intervention. Persistent failures accounted for roughly 22%: schema validation rejections, unresolvable account references, and transaction IDs whose corresponding records had already settled through a separate processing path.

Records in the persistent failure class cannot be retried — retrying them produces the same rejection. A compensating payment-failed event on the command topic is what routes them correctly: it tells the originating service the transaction is terminal and generates an operations alert for manual review. Infrastructure-class failures, around 5% of DLQ volume, cleared through Kubernetes health-check-triggered consumer restarts with no DLQ intervention needed.

The classification logic uses an error-class header that the retry framework attaches before routing to the DLQ. Transient-class messages get re-enqueued on the main topic with an incremented retry counter and an exponential backoff header. Persistent-class messages get the compensating event path. Odofin et al. (2024) showed in a Kafka-Camunda BPM integration that this header-driven classification can coordinate DLQ routing and saga compensation without a separate distributed transaction coordinator — the same principle the 4PF uses here.

5.3 PSD2 Compliance Architecture

Payment record retention under PSD2 Article 32 has two dimensions: duration and tamper evidence. On the duration side, five years of records must be kept. On the tamper evidence side, those records must be demonstrably unmodified. Kafka's log structure handles tamper evidence without extra infrastructure because written offsets cannot be modified in place — the audit trail is built into the storage model. Log compaction with cleanup.policy=compact+delete retains the latest record per payment key while the retention ceiling manages storage growth, giving compliance auditors both current state snapshots and the historical event sequence needed for dispute resolution.

Schema registry enforcement closes the chain of custody for historical deserialization. Every payment event carries a schema ID header resolvable against the registry snapshot active at the time of original publication. An auditor replaying the payment-events topic five years after initial publication can deserialize every record using the schema version in effect at the time of the original write, even if production schemas have evolved through multiple versions in the intervening period. Adari (2024) established the open banking API interoperability requirements governing the cross-institution payment flows the 4PF handles; Gipp (2026) documented the resilient middleware architecture requirements for open banking platforms that complement Pattern 4's event sourcing audit approach.

6. Discussion

The 4PF's deployment sequence puts Pattern 2 first because consumer restarts are operationally more frequent than producer network failures by a wide margin. An institution deploying only Pattern 2 gets immediate, measurable protection against the most common failure mode with no throughput cost. Adding Pattern 1's transactional producer introduces a 10 to 15 percent throughput reduction on EOS-scoped topics; that cost is worth accepting after Pattern 2 is stable, not before.

Three boundaries limit where the 4PF applies. It is specified and validated exclusively for Apache Kafka and Spring Boot. Institutions on AWS Kinesis, Azure Event Hubs, or Google Cloud Pub/Sub would need to map the four patterns to their platform's analogous primitives. The mapping is not always one-to-one: Azure Event Hubs does not expose a transactional producer API equivalent; AWS Kinesis handles exactly-once semantics at the shard level rather than the partition level. The 4PF does not address those platform-specific cases.

The Redis deduplication cache in Pattern 2 introduces a cache availability dependency. If Redis becomes unavailable, the fingerprint check fails and the consumer cannot safely determine whether a record was already processed. Both reference deployments mitigated this with Redis Sentinel replication, but the 4PF specification does not mandate a specific cache topology; practitioners must make that infrastructure decision based on their availability budget. Pattern 4's CQRS separation adds two topic families, two consumer group configurations, and two schema evolution tracks to the operational surface area.

Microservice scaling under variable load is an ongoing challenge in enterprise financial systems, as Suleiman and Murtaza (2024) documented; Pattern 4 compounds that challenge by adding CQRS operational overhead on top of the scaling work already required for Patterns 1 through 3. Institutions whose compliance requirements do not extend beyond a one-year retention window may find Pattern 4's cost-benefit ratio unfavorable. The framework is designed to accommodate partial adoption — the first three patterns deliver meaningful reliability improvements without Pattern 4.

The practical value of the 4PF is not that any individual pattern is novel. Producer EOS, consumer idempotency, manual offset commit, and CQRS are each documented independently in the literature. The value is in the specification of their interaction and the sequencing of their deployment. George (2022) supplies the multi-cloud infrastructure context; Deshpande (2025) validates the Spring Boot implementation path; Chirukuri (2025) furnishes the production throughput evidence. The 4PF synthesizes those independent contributions into a deployment guide that financial engineering teams can follow without independently discovering the interdependencies between the four components.

7. Conclusion

A Kafka deployment that activates idempotent producers but skips consumer-layer deduplication is not an exactly-once system. Neither is one that implements manual offset commit but leaves the audit record coupled to the query model. Exactly-once processing in a financial context requires all four of the 4PF's patterns operating together, each closing a failure mode the others cannot reach.

Pattern 1 addresses broker-side write duplication. Pattern 2 addresses application-layer reprocessing after consumer restarts. Pattern 3 eliminates silent message loss from timer-based offset commit. Pattern 4

separates the immutable audit record from the operational query path and enforces the five-year retention mandate PSD2 requires. Production deployments using all four patterns achieved sub-five-second settlement latency with zero duplicate payment events across the measured deployment windows.

For financial institutions planning Kafka adoption, the sequencing guidance is the most immediately actionable part of this framework. Deploy Pattern 2 first: it costs nothing in throughput and closes the highest-frequency failure mode. Then deploy Pattern 1. Then Pattern 3. Deploy Pattern 4 only after the first three are stable and the compliance record requirements are confirmed. That sequence makes each deployment phase independently valuable and avoids the common mistake of treating Kafka EOS as a single configuration toggle rather than a four-component reliability system.

References

1. Adari, V. K. (2024). APIs and open banking: Driving interoperability in the financial sector. International Journal of Research in Computer Applications and Information Technology (IJRCAIT). https://doi.org/10.5281/zenodo.14283143
2. Bellemare, A. (2020). Building event-driven microservices: Leveraging organizational data at scale. O'Reilly Media. https://www.oreilly.com/library/view/building-event-driven-microservices/9781492057888/
3. Chirukuri, V. G. K. (2025). Building an end-to-end reconciliation platform for accurate B2B payments in new-age fintech using Apache Kafka. European Journal of Computer Science and Information Technology (EJCSIT), 13(4), 54-70. https://eajournals.org/ejcsit/vol13-issue-4-2025/building-an-end-to-end-reconciliation-platform-for-accurate-b2b-payments-in-new-age-fintech-distributed-ecosystems-a-case-study-using-microservices-and-kafka/
4. Dayarathna, M., & Perera, S. (2018). Recent advancements in event processing. ACM Computing Surveys, 51(2), 33:1-33:36. https://doi.org/10.1145/3170432
5. Deshpande, R. A. (2025). Application of Spring Boot microservice architecture for scaling banking applications. The American Journal of Engineering and Technology (TAJET), 7(9). https://doi.org/10.37547/tajet/Volume07Issue09-09
6. George, J. (2022). Optimizing hybrid and multi-cloud architectures for real-time data streaming and analytics. World Journal of Advanced Engineering Technology and Sciences (WJAETS), 7(1). https://doi.org/10.30574/wjaets.2022.7.1.0087
7. Gipp, B. (2026). Resilient middleware and API-driven interoperable platforms for open banking and large-scale enterprise systems. International Journal of Research in AI (IJRAI), 9(2). https://ijrai.org/index.php/ijrai/article/view/442
8. Hebbar, K. S. (2025). Optimizing distributed transactions in banking APIs: Saga pattern vs. two-phase commit. The American Journal of Engineering and Technology (TAJET), 7(6). https://doi.org/10.37547/tajet/Volume07Issue06-18
9. Kennady, V., & Mayilsamy, P. (2022). Migration of batch processing systems in financial sectors to near real-time processing using Apache Kafka. International Journal of Scientific Research and Publications (IJSRP). https://www.ijsrp.org/research-paper-0722/ijsrp-p12755.pdf
10. Kleppmann, M. (2017). Designing data-intensive applications. O'Reilly Media. https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/
11. Laigner, R., et al. (2024). An empirical study on challenges of event management in microservice architectures. ResearchGate. https://www.researchgate.net/publication/385036085
12. Mondal, S., et al. (2023). Kubernetes autoscaling for microservices. MDPI Mathematics, 11(12), 2675. https://doi.org/10.3390/math11122675
13. Narayanan, A. (2025). Optimizing event-driven architectures for real-time financial transactions. World Journal of Advanced Engineering Technology and Sciences (WJAETS), 15(1). https://doi.org/10.30574/wjaets.2025.15.1.0199
14. Odofin, O. T., et al. (2024). Designing event-driven architecture for financial systems using Kafka, Camunda BPM, and programmatic transaction management. International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET). https://doi.org/10.32628/IJSRSET25121178
15. Overeem, M., Spoor, M., & Jansen, S. (2017). The dark side of event sourcing: Managing data conversion. IEEE SANER 2017, pp. 420-429. https://ieeexplore.ieee.org/document/7884621

16. Pallaprolu, S. (2025). Event-driven architecture: A modern paradigm for real-time responsive systems. World Journal of Advanced Engineering Technology and Sciences (WJAETS), 15(2). https://doi.org/10.30574/wjaets.2025.15.2.0826
17. Raptis, T. P., & Passarella, A. (2023). A survey on networked data streaming with Apache Kafka. IEEE Access, 11, 85333-85350. https://doi.org/10.1109/ACCESS.2023.3303810
18. Nadia Suleiman, & Yusuf Murtaza. (2024). Scaling Microservices for Enterprise Applications: Comprehensive Strategies for Achieving High Availability, Performance Optimization, Resilience, and Seamless Integration in Large-Scale Distributed Systems and Complex Cloud Environments. Applied Research in Artificial Intelligence and Cloud Computing, 7(6), 46–82. Retrieved from https://researchberg.com/index.php/araic/article/view/208
19. Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: A systematic mapping study. In Proceedings of CLOSER 2018, pp. 221-232. https://www.scitepress.org/papers/2018/67983/67983.pdf
20. Vallabhaneni, S. (2025). Demystifying event-driven architecture in modern distributed systems. World Journal of Advanced Engineering Technology and Sciences (WJAETS), 15(1). https://doi.org/10.30574/wjaets.2025.15.1.0402
21. Venkatachalapathi, K. (2024). Event-driven architecture: Harnessing Kafka and Spring Boot for scalable, real-time applications. International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT). https://doi.org/10.32628/CSEIT24106193
22. Yadlapalli, R. (2025). Technical review: Real-time payment processing system for banking industry. World Journal of Advanced Engineering Technology and Sciences (WJAETS), 15(3). https://doi.org/10.30574/wjaets.2025.15.3.1147