International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Designing a Scalable Event-Driven Mortgage Servicing Platform for Deterministic Financial Processing

Gowri Shankar Chaganti

Abstract: Existing mortgage servicing systems rely on batch processing, monolithic business logic, and inflexible infrastructure models that do not scale to the volume and regulatory complexity of modern financial services. This article describes a deterministic, event-driven architecture in which each state change of a loan is expressed as an immutable and replayable event. Key architectural principles include stateless processing services, externalized and versioned business rules, and partitioned event streams for horizontal scalability. The architecture also incorporates a multi-layer idempotency framework that ensures exactly-once processing semantics across financial ledgers. Financial and procedural compliance audit trails are incorporated into the design through integrated workflows and consolidated within the same event log in an append-only manner for regulatory traceability. Compared to existing batch systems, the comparative evaluation demonstrates reduced latency from hours in batch responses to near real-time, near-linear scalability with additional processing nodes, and no observable divergence in financial outputs when replayed in controlled conditions, showing that deterministic event-driven processing is technically feasible on production mortgage servicing infrastructure and yields operational benefits in key areas including scalability, auditability, and regulatory responsiveness

Keywords: *Event-Driven Architecture, Mortgage Servicing Platform, Deterministic Financial Processing, Regulatory Compliance Automation, Distributed Ledger Systems*

1. Introduction

Mortgage servicing is among the most operationally complex functions within financial institutions. At its core, it involves the accurate and timely processing of borrower payments, escrow management, interest calculations, delinquency handling, and a growing volume of regulatory reporting obligations [9, 22]. Each function must be performed precisely, or it will lead to financial and legal consequences. The systems that support these operations have historically been mainframe-based platforms built around nightly batch processing cycles and monolithic business logic [1]. These systems worked well for the situations they were made for, but they are becoming less and less able to handle the needs of today's mortgage servicing environments. The central problem is structural. Legacy platforms process data in batches, embed business rules deep within their codebases, and scale only through expensive hardware upgrades [4]. As regulatory frameworks have grown more detailed, as consumer protection requirements have expanded under instruments such as Regulation X and Regulation Z, and as mortgage portfolios have grown in scale and complexity, the gap between what these systems can deliver and what the business actually needs has widened significantly [7, 23]. This article addresses that gap by examining the design of a scalable, event-driven mortgage servicing platform. It covers the limitations of legacy systems in detail, then builds the case for an event-driven architectural model through its core design principles, its approach to deterministic financial processing, its auditability and compliance capabilities, its scalability architecture, and its support for dynamic business rule management. The article concludes by examining the economic and operational path from legacy infrastructure to a modern, variable-cost servicing platform.

1.1 Contributions

This paper makes the following contributions:

1. A deterministic event-driven financial processing model enabling replayable and consistent loan state computation
2. A multi-layer idempotency framework ensuring exactly-once financial ledger semantics
3. A unified auditability architecture integrating financial and procedural compliance traceability

Independent Researcher, USA

4. A scalable partitioned event-processing topology tailored for high-volume mortgage servicing workloads

2. Structural Failures of Legacy Mortgage Accounting Systems

2.1 Batch Processing and Its Operational Consequences

Most legacy mortgage accounting systems also used a batch processing model, such as amassing transactions during a business day and reconciling during nightly batch jobs. Payments, interest accruals, and changes to escrow balances are applied in batches rather than in real time [1]. While this might have been feasible when hardware was expensive and volumes were relatively low, it is far less suited to the operational scale of modern mortgage servicing. The most direct consequence is data latency. A borrower payment received at midday does not affect the loan's outstanding principal or escrow balance until the overnight batch run completes. Any system that depends on current loan state, including customer inquiry tools, delinquency monitors, or loss-mitigation workflows, is therefore working with outdated information for a significant portion of the day [4]. In high-volume environments, this lag compounds. Secondary batch jobs are sometimes introduced to partially reconcile intra-day activity, but these workarounds add operational complexity without resolving the fundamental delay [20]. Batch architectures also introduce risks at the boundaries of processing. At any point in the middle of a nightly batch run, the loan portfolio may be in a partially updated state. Defining the exact restart point is difficult when there are millions of dependent loans, making redoing part of the batch the most practical way to recover. The longer the iteration, the more likely it will fail [1].

2.2 Rigidity of Business Logic and the Cost of Regulatory Change

Beyond processing latency, legacy systems impose a structural rigidity that makes regulatory adaptation expensive. In these environments, developers typically embed business rules within procedural code, tightly coupling data models and output formatting. Changing a rule requires navigating deep dependencies within the application layer [3]. This process is expensive, as development teams have to identify the right logic in a codebase that was rarely designed with modularity in mind, make the change, and run multiple weeks of regression testing. This endemic risk of indirect modification of other processing paths in tightly coupled systems means that each update needs to go through exhaustive testing (even for a nominal change) [7], which is fundamentally incompatible with the new pace of change of mortgage servicing regulations [9]. The CFPB's mortgage servicing rules under Regulation X and Regulation Z, for example, specify detailed requirements for payment processing timelines, escrow analysis disclosures, and loss-mitigation procedures [23]. Meeting these requirements in a legacy system often demands large development cycles that consume institutional resources and introduce deployment risk across the entire servicing platform [22].

2.3 Scalability Ceilings and Infrastructure Inflexibility

Legacy mainframe environments scale vertically. Upgrading the host system requires additional capacity rather than distributing the workload across additional nodes. This architectural ceiling becomes a real operational problem as mortgage portfolios grow [4]. Processing windows that were adequate for one portfolio size may become untenable as volumes double or triple through acquisition or organic growth. The infrastructure cost model compounds this problem. Mainframe environments require consistent capital expenditure regardless of actual processing demand. Peak capacity must be provisioned to handle month-end or year-end event surges, which means the institution pays for that capacity during the far more common periods of lower utilization [8]. This mismatch between fixed infrastructure cost and variable processing demand creates financial inefficiency and limits the institution's ability to absorb portfolio growth without proportionally large infrastructure investments [18].

3. Architectural Design of the Mortgage Servicing Event Platform

3.1 Core Architectural Principles

This paper proposes an architectural model that departs from legacy batch-based systems by inverting their fundamental assumptions. Instead of representing loan state as a mutable record incrementally updated through batch processes, the proposed model represents each loan state transition as an immutable event appended to a persistent event log. Within this framework, the state of a loan account at any point is computed as a deterministic function of the ordered sequence of events applied

to it [19]. This formulation enables inherent auditability, replayability, and deterministic financial processing, which are critical requirements for modern mortgage servicing platforms.

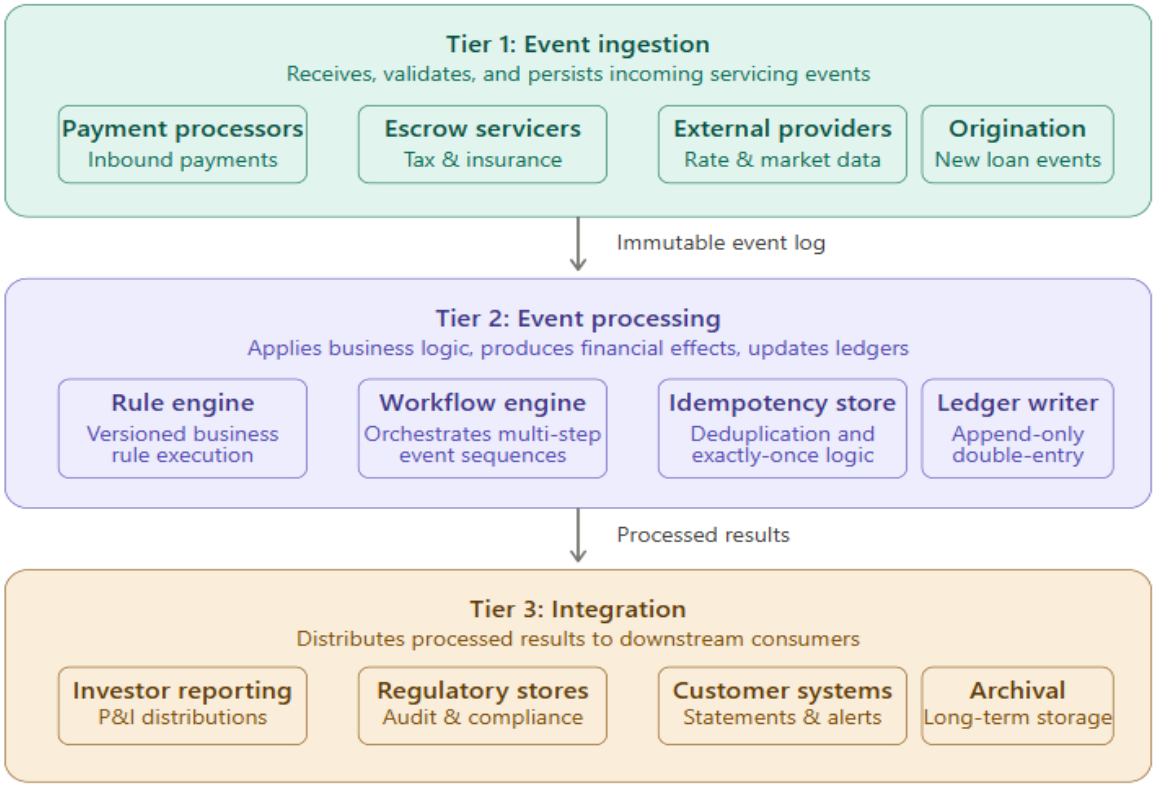

Figure 2: Three-Tier Architecture of the Mortgage Servicing Event Platform [2, 4]

The three tiers are (1) the event ingestion tier, which receives, validates, and persists servicing events from payment processors, escrow servicers, and external data providers; (2) the event processing tier, which applies business rules to create the financial impact, performs ledger updates, and sends the results to investor reporting systems and regulatory data stores; and (3) customer-facing applications via the integration tier (as shown in figure 2) [2]. Each tier can be deployed and scaled independently. Coupling between the tiers is achieved through event schemas. These concerns must be separated if changes to the processing logic are to be decoupled from any changes to ingestion or integration behavior in a regulatory environment [20].

3.2 Event Taxonomy and Lifecycle Modeling

To maintain processing integrity, event modeling must be precise because not every loan state transition represents an event. Payment events have value and allocation logic. Examples of externally triggered escrow events are tax authority disbursements and insurance premiums. Delinquency events are transitions that start regulatory notification workflows. Modification events alter fundamental terms of the loan, which will also require recalculation of amortization schedules [1]. The events have a schema that is specific to their type. An event object contains an event identifier, a loan identifier, the current time, the time at which the event is being persisted, an event type, and an event payload to be processed. Schemas have versioning to address backwards compatibility issues as the platform evolves [4]. Once an event has occurred, it is strictly immutable, so corrections are represented as new compensating events that are causally related [6].

Event category	Subtype	Triggering condition	Required payload fields	Downstream processing targets
Payment	Standard payment	The borrower remits scheduled amount	Event ID, loan ID, amount, payment date, channel, received timestamp	P&I allocation service, escrow service, ledger writer, investor reporting
	Partial payment	Amount less than scheduled due	Event ID, loan ID, amount, partial flag, shortage amount	Delinquency monitor, suspense account, compliance workflow
Escrow	Tax disbursement	Tax authority payment due date reached	Event ID, loan ID, tax year, disbursement amount, payee ID	Escrow sub-account, ledger writer, escrow analysis service
	Escrow analysis	Annual escrow review cycle	Event ID, loan ID, analysis period, projected balance, shortage or surplus	Disclosure generator, payment adjustment service, regulatory store
Delinquency	Missed payment	Payment not received by due date	Event ID, loan ID, missed due date, days past due, current balance	Compliance workflow engine, loss mitigation service, investor reporting
	Late fee assessment	The grace period expired	Event ID, loan ID, fee amount, assessment date, rule version	Ledger writer, customer notification service
Correction	Compensating event	Processing error identified in prior event	Event ID, loan ID, original event ID reference, correction payload, authorised-by timestamp	Reversal engine, ledger writer, reconciliation service, audit log

Table 1: Mortgage Servicing Event Taxonomy [3, 9]

This immutability requirement is not just something that makes sense to implement for design purposes; it is a requirement in order to attest, during a regulatory audit, exactly what data was available at the time a financial calculation was performed [9].

3.3 Processing Topology and Workflow Orchestration

The event processing also follows a strict topology for achieving determinism and isolation: Each event type is assigned to a processing service that is guaranteed to run the defined financial logic for that event type. The services are stateless. That is, the state needed to process the event is either embedded in the event or fetched from a lone state store at the start of processing the event [4]. This means that reprocessing an event after a failure has the same result as it had originally. Pat Helland establishes that systems operating beyond transaction boundaries must explicitly design for deferred consistency, reinforcing the principle that stateless, event-driven processing services eliminate the cross-service coordination overhead that makes legacy monolithic workflows brittle [24].

Workflow orchestration defines the order in which the steps will be executed because they depend on each other's results. In a payment, for example, the steps could include payment validation, allocation of principal and interest, escrow disbursement, and posting to the ledger. The orchestration layer provides stateful in-flight workflows, as well as retry, timeout, and compensation logic for failed steps [2]. It enables the platform to decouple state from individual services. This frees the platform from the tight coupling of business logic and process management that is the defining weakness of legacy monolithic systems [3].

3.4 Comparison with Existing Architectures

This section contrasts the proposed architecture with alternative approaches to highlight its structural advantages. These contrasts will clarify the problem that the proposed architecture addresses and the specific capabilities that it offers compared to other architectures.

3.4.1 Traditional Batch-Processing Architectures

Batch-processing systems are the main type used in legacy mortgage servicing. In such systems, loan state transition updates that happen over a processing window (e.g., overnight) are applied when the run is scheduled to occur. The most important constraint is time: for a loan, as illustrated in any dependent system like delinquency monitors, the customer inquiry system, or the pipeline reporting to investors, it is only updated by the last batch run. This intra-day information lag is not just

a nuisance of latency but has regulatory implications [1, 4]. If Regulation X, for example, says that certain delinquency notifications must happen if a certain payment status is reached [9], then a system that updates the state of the loan only once every twenty-four hours may miss the opportunity to perform certain workflows in time, resulting in a compliance exposure that is an artifact of architecture, not intent. The proposed event-driven model solves this problem at a structural level: as all loan state changes are modeled as events and processed in near-real-time, the current state of the loan can always be derived from the up-to-date event log. This is not only an operational improvement: downstream systems that consume the event stream are presented with state changes as they happen, not in the granularity of batch cycles. This changes the compliance risk profile of the platform, as it removes the processing delay that makes the Regulation X timeline probabilistic in batch systems [9, 20].

3.4.2 Microservices Architectures Without Event Sourcing

Microservices architectures, which decompose processing logic into independently deployable services, are widely adopted in the financial technology sector as a migration path from monolithic legacy systems. However, the adoption of a microservices topology does not in itself impart deterministic replayability or consistent financial computation. In the absence of event sourcing, each microservice has its own state store and typically communicates via synchronous API calls or a loosely organized message queue. The resulting loan account state is distributed across the state stores controlled by each service, creating an intrinsic auditability problem. The reconstruction of the chain of operations that led to any state requires the cross-service correlation of records that may lack a unique transaction ID or ordering guarantees [4].

This architectural choice is not practical in a regulated mortgage servicing environment. If the regulator asks the institution to explain exactly how an escrow adjustment happened or how a delinquency classification was made, without event sourcing, the answer may need to be constructed from disparate logs in a microservices architecture. This reconstruction is lossy by definition because it loses intermediate states that occur only in memory during the synchronous calls. The proposed architecture addresses this issue by requiring every service to emit a structured and immutable event record containing the full input payload, the applied transformation, and the resulting output, regardless of whether it is stateless or stateful [4, 6]. This means that the event log is the single source of truth for the entire pipeline, and the audit record is a structural property rather than a forensic artifact reconstituted after the fact.

3.4.3 Data Lake-Based Processing Pipelines

Data lake architectures are increasingly popular in financial services organizations for aggregating large volumes of structured and unstructured data from the operational environment and optimizing analytics, reporting, and machine learning workloads. In mortgage servicing, some companies use data lake pipelines to extract loan data from each system of record into a central analytical data store to enable regulatory reporting and portfolio analysis without modifying the upstream systems of record. While this has merit for analytical use cases, it is not architecturally suited for mortgage servicing operations for two reasons.

First, many data lake pipelines are designed for eventual consistency, with records from operational stores ingested in batches and micro-batches, and the view of any given loan account provided by the data lake is likely to be stale for minutes, hours, or longer, depending on ingest cadence [8]. In addition, there is a time lag for any reporting where point-in-time accuracy is important, such as month-end investor remittances or escrow analysis disclosures. This creates a reconciliation burden and an operational risk where reports are run against a loan that is not in exactly the same state as in the operational record. More importantly, data lake pipelines do not guarantee transactional integrity across the loan lifecycle. The lake cannot provide exactly-once processing semantics, which are required by a financial ledger, because it is only downstream from operational data and not the system of record. If a payment event is ingested twice due to a failure in the pipeline, then there will be a duplicate balance in the lake, with no automatic mechanism to detect or correct it [5, 12].

The proposed architecture systematically inverts this dependency such that the event log is the source of truth, which gives rise to all other views of the system (ledger projections, regulatory reports, investor summaries, etc.) via replay or query. This way, the relationship between these secondary views and the event log, backed by a concrete sequence of immutable events, is such that discrepancies are projection errors rather than data quality problems.

The guarantees of the idempotency scheme from Section 4.2 hold for all consumers of the event stream. This level of consistency is impossible for other data lake designs that rely on ingested copies [4, 6].

4. Deterministic Financial Processing and Accounting Integrity

4.1 Defining Determinism in Financial Computation

Determinism in financial processing means that the same inputs will produce the same outputs for the same interpretation across all executions, with special constraints in mortgage accounting. That is, calculations of interest accrual shall yield the same result, whether computed on the primary node of a server or a failover replica of that server. The amortization schedules must be the same. The escrow shortage calculations cannot depend on the order of receiving events in the escrow [4]. Achieving determinism requires deliberate decisions at multiple levels. For all financial computations, replace floating-point arithmetic, known for producing platform-dependent rounding behavior, with fixed-point or arbitrary-precision decimal arithmetic [16]. Day-count conventions, which vary across loan product types, must be explicitly encoded in the event payload rather than inferred from the system state. Processing timestamps must reference a canonical event time recorded at ingestion rather than the wall-clock time of the processing node. Each of these decisions eliminates a potential source of inconsistency [19].

Precision and the definition of a rounding function affect numerical accuracy in the IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019) [16]. Mortgage servicing platforms that perform calculations using numbers without a well-defined standard of arithmetic risk subtle differences that may not be detected until the point of reconciliation or audit [4].

4.2 Idempotency Mechanisms and Duplicate Event Handling

Distributed processing environments can deliver duplicate events. Network retries, message broker redeliveries, and failover scenarios can all cause a processing service to receive the same event more than once. Without explicit idempotency controls, duplicate processing would produce double-posted ledger entries, inflated escrow balances, and incorrect amortization states [5]. The platform implements multi-layer idempotency. Each event has a globally unique identifier that is assigned at the point of ingestion. Every processing service has a registry of idempotency states, which tracks the result of the processing of each event identifier. In practice, before executing the logic necessary to process an event, the service checks the idempotency registry to see if the event has already been processed and returns the stored result [2]. The registry entry is written transactionally together with the processing result to avoid ambiguous states in the event of partial failures [4].

This mechanism also supports exactly-once processing semantics in the ledger posting layer. Financial ledger entries are associated with event identifiers, and the ledger system enforces uniqueness constraints on those identifiers. Any attempt to post a duplicate entry is rejected at the database layer, providing a final safety net independent of application-level logic [12].

4.3 Ledger Architecture and Financial Reconciliation

The financial basis of the system is an append-only double-entry bookkeeping system. Each post-processing transaction results in a series of balanced debits/credits keyed on the originating event ID, the loan account ID, the transaction date, and a set of general ledger account codes. The ledger never modifies an entry in place but instead appends a reverse entry and a corrected entry, both identified by the original event identifier [6]. Reconciliation processes are performed against the event stream, not the ledger: the reconciliation service replays a bounded number of events to compute the expected ledger state, which is compared against the posted balances [15]. Discrepancies, then, are not silent inconsistencies but actionable exceptions: because the event log is the definitive record of what happened, while the ledger is a projection upon it, the event log wins in case of any contradictions [4].

Helland further argues that immutability is not merely a design convenience but a foundational property that makes distributed data systems auditable and recoverable by construction, a principle the append-only ledger architecture directly embodies [25].

The append-only design also simplifies disaster recovery. Because the ledger is a derivable projection of the event log, a corrupted or lost ledger can be fully reconstructed by replaying the event history. This property, highlighted in the event sourcing literature, provides a level of recoverability

that periodic-snapshot-based systems cannot match [19].

5. Auditability, Regulatory Compliance, and Traceability

5.1 Event Logs as Regulatory Evidence

Mortgage servicing is usually subject to regulations that extend beyond merely auditing. Servicers need to be able to show what inputs, rules, and calculations led to any result. In the United States, the CFPB's Regulation X requires a servicer to maintain and make available documentation of the methods used to apply payments and perform escrow analyses [9]. Regulation Z requires similar documentation for interest calculations and disclosures [23]. Investor agreements also contain details of how principal, interest, and fees are allocated.

The event log satisfies these requirements through structural completeness. Every servicing action is represented as a persisted event with an immutable record of its payload, its processing timestamp, the version of the business rule applied, and the resulting financial effects [15]. The log is not a summary or a periodically updated snapshot. It is a granular, ordered record from which the full history of any loan account can be reconstructed. Regulatory examiners receive a deterministically computed answer about the state of a specific loan as of a specific date, which is derived directly from the event sequence [19].

5.2 Temporal Query and State Reconstruction

One of the more consequential properties of the event sourcing model is its support for temporal queries. The state of a loan account at any point in its history is computable by replaying the event log up to the timestamp in question [4]. This capability eliminates a significant compliance gap that exists in legacy systems, where historical states are represented only through periodic balance snapshots, and the logic used to produce those snapshots may no longer be reproducible. For temporal queries, the system uses a snapshot optimization by periodically committing a checkpoint with the loan state at that point in time. For any queries after the most recent checkpoint, the system replays only events that happen after that time to save on time and space while providing the same output [6]. Checkpoints are immutable and can be recomputed from the event log in the event that they are corrupted [19].

5.3 Compliance Workflow Integration and Notification Traceability

Procedural timing requirements for delinquency notices, loss mitigation offers, and escrow analysis disclosures are set forth in Regulation X [9] and not necessarily completely dependent upon the accuracy of the mortgage servicing financials. Regulatory penalties may be imposed for procedural violations notwithstanding the accuracy of the accounting for mortgage servicing activities [22].

The platform integrates compliance workflows directly into the event processing topology. When a delinquency event crosses a defined threshold, the event processor simultaneously updates the loan state and publishes a compliance trigger event to the workflow engine. The workflow engine initiates the required notification sequence, records each step with a timestamp and delivery confirmation, and persists the complete procedural record against the loan identifier [2]. The result is a unified audit trail that covers both financial and procedural servicing activity [20]. The importance of this unified trail extends to examination scenarios. As noted in the CFPB's assessment of mortgage servicing rules, institutions must be able to demonstrate procedural compliance with specific notification and review timelines, not merely assert that those procedures occurred [22]. An event-driven audit trail directly addresses this requirement by capturing both the financial and procedural record within the same immutable log structure [23].

6. Scalability Architecture for Large Mortgage Portfolios

6.1 Event Partitioning and Parallel Processing

To support the processing of millions of loan events per day while meeting latency and ordering constraints, the platform partitions the event stream by loan identifier, ensuring that all events for a given loan are routed to the same partition and processed in order, without enforcing global ordering guarantees across all loans [5]. This allows a single consumer to read events from each partition in order. A coordination service tracks active nodes, reassigning partitions from failed nodes to new active nodes as they come online and offline. It performs partitioning on a pool of processing nodes and scales nearly linearly due to its architecture. The addition of processing nodes increases throughput in a linear manner across the range of portfolio sizes observed in institutional mortgage servicing operations.

The Kafka distributed messaging model demonstrates that log-based partitioning with consumer group coordination provides both high throughput and durable message delivery [5]. These properties are directly applicable to the context of mortgage event processing, where durability and ordering are non-negotiable requirements. Thomson et al. demonstrate that deterministic transaction ordering across partitioned database systems can be achieved without distributed locking by pre-ordering inputs before execution, a result that supports the platform's partition-by-loan-identifier strategy as a means of preserving causal ordering without sacrificing throughput [26].

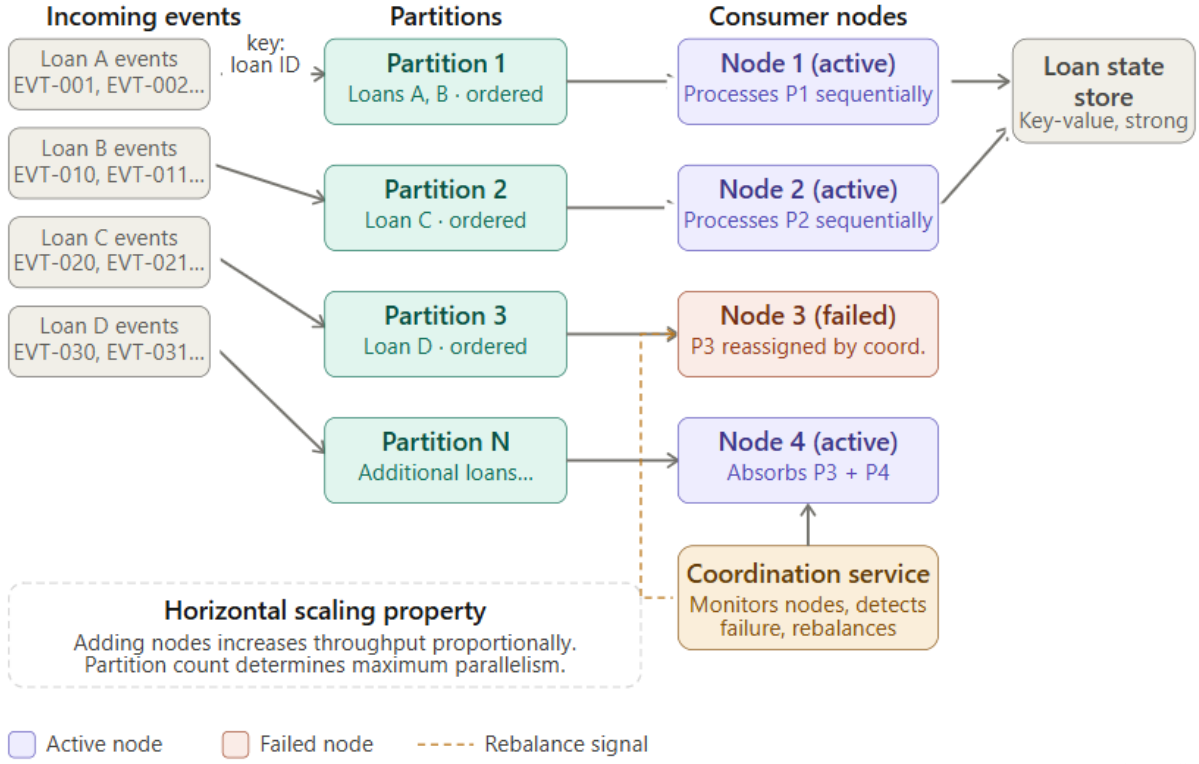

Figure 3: Event Stream Partitioning Architecture [5, 12, 13]

6.2 State Store Design for Distributed Processing

Stateless processing services require access to the current loan state at the outset of each event processing cycle. This state must be read consistently, updated atomically with processing results, and available with latency characteristics that do not become the bottleneck in a high-throughput pipeline. A distributed key-value store configured with strong consistency semantics and optimized for read-heavy access patterns provides these properties [12].

Loan state records are structured to contain all data elements required by any processing service, eliminating the need for joins or multi-record reads during processing. The state store is located next to processing nodes when the infrastructure allows it, which cuts down on network round-trip latency for the most common access pattern [12]. Write operations are batched transactionally with idempotency registry updates, ensuring that state and deduplication records remain synchronized under failure and retry scenarios [4]. Considering the state store, Lee et al. point out that, for distributed systems, there is a tension between consistency and availability and that heterogeneous systems may have different layers that require different consistency levels [13]. The financial ledger state requires strong consistency. Eventual consistency is acceptable for derived views in reporting or inquiry systems, provided the platform has an opportunity to optimize for read-heavy workloads without compromising the integrity of the primary data-processing path.

6.3 Stream Processing Frameworks and Throughput Management

The event processing tier can also benefit from the properties of modern stream processing systems that support high throughput, low latency, and high fault-tolerance guarantees. Apache Flink provides stream processing and batch processing as a single

processing engine, allowing real-time processing of events and periodic reconciliation of batches to run on the same platform [10]. Apache Spark's Structured Streaming is a declarative API to implement streaming applications in a fault-tolerant and scalable way [12]. There are also many other factors that cause spikes of additional work many times larger than the baseline work, including month-end payment cycles, annual escrow analysis cycles, and tax payment disbursement cycles. Because an infrastructure model that is provisioned to the peak capacity incurs unnecessary costs at off-peak times [8], the stream processing tier is superficially horizontally capacity elastic on the throughput dimension: the system adds more processing nodes to the consumer group when the queue depth crosses certain thresholds. Once throughput stabilizes at normal levels, excess capacity is released, and the entire scaling cycle is completed without any service interruption [20].

Framework	Consistency model	Fault tolerance mechanism	Scaling approach	Applicable processing stage
Apache Kafka (log broker)	At-least-once delivery: once with idempotent producers and transactional consumers	Partition replication across broker nodes; leader election on failure	Horizontal: add partitions and consumer group members; near-linear throughput scaling	Event ingestion tier: durable ordered log for all incoming servicing events
Apache Flink	Exactly once via distributed snapshots (Chandy-Lamport algorithm)	Periodic checkpoints to distributed storage; automatic job restart from last checkpoint	Horizontal: parallelism configured per operator; dynamic resource scaling with reactive mode	Event processing tier: stateful stream computations, amortisation schedules, real-time accrual
Apache Spark Structured Streaming	Exactly once via write-ahead logs and idempotent sinks	Driver and executor restart; state store recovery from checkpoint directory	Horizontal: executor pool scales with cluster manager (YARN, Kubernetes); micro-batch or continuous mode	Batch reconciliation layer: periodic ledger validation replays, escrow analysis aggregations
Google Spanner (state store)	External consistency (stronger than serializable): globally consistent reads without locking	Paxos-based replication across zones; automatic failover with no data loss	Horizontal: split-based auto-sharding; throughput scales with replica count and node size	Loan state store: current loan state reads at processing start; atomic writes with idempotency registry

Table 2: Stream Processing Framework Properties Relevant to Mortgage Event Processing [5, 11, 12, 13]

7. Dynamic Business Rule Management and Platform Extensibility

7.1 Separating Rules from the Processing Engine

The fragility of rule changes in legacy systems stems from the co-location of business logic with processing infrastructure. When rule definitions reside in the same codebase as the engine that executes them, any modification carries risk to the entire system. The modern platform resolves the issue of treating business rules as externalized, independently versioned artifacts [3]. Rules are defined, versioned, and deployed independently from the processing services that invoke them. This separation is implemented through a rule engine that processes loan events against a configurable rule set loaded at the outset of each processing cycle. Rule definitions are stored in a versioned repository with a full change history. Each loan account carries a reference to the specific rule version applicable to its product type and origination date [14]. A regulatory change affecting one product category can be deployed and activated for the relevant loans without touching the processing infrastructure or the rules governing other product categories. Jin et al. demonstrate how dynamic rule inference in edge computing environments benefits

from precisely this kind of externalized, configurable rule management, a principle that applies equally well to financial processing platforms [14].

7.2 Rule Testing, Validation, and Controlled Rollout

The ability to deploy rules independently creates a new category of operational risk. A rule that is logically consistent but economically incorrect can produce systematic errors across every loan to which it is applied. The platform reduces this risk by using a structured rule validation pipeline. New rule versions are evaluated against a library of reference test cases, including edge cases drawn from historical processing exceptions [1]. Rules that produce unexpected results against reference cases are rejected before deployment.

For rules that pass checking, shadow processing allows a new version to be run alongside the deployed version. The new version runs against a stream of event data, and the outcomes are compared against the deployed version's outcomes, without affecting the posted ledger entries. Any divergences will be pointed out to the rule designers so that any unexpected differences in behavior in the new version are identified before the new, updated version is released to production. This makes changing regulatory rules much lower risk. It also provides legacy systems a way to test changes that their tightly coupled architectures cannot accommodate [3].

7.3 Extending the Platform with New Event Types

Mortgage product innovation and regulatory evolution periodically create servicing scenarios that the original platform design did not anticipate. The ability to extend the event taxonomy without disrupting existing processing pipelines is a necessary property of any durable platform architecture [4]. New event types can be created by defining a new event schema and publishing it to the schema registry. A subscription manifest declares what event types processing services can handle. In this way new processing services can be deployed and designed to respond to new event types without changing services already deployed [2]. The event routing layer uses the service subscription manifest to route the events it receives to the appropriate processing services, allowing the routing logic to be extensible as the platform extends its capabilities. This model is based on the domain-driven design principle that each subdomain is able to evolve separately in a bounded context [3].

8. Transition to a Variable-Cost Mortgage Servicing Platform

8.1 Economic Architecture of Legacy Infrastructure

Legacy mainframe servicing environments have a capital-intensive cost structure that requires organizations to acquire physical hardware capacity well in advance of demand. License costs, hardware maintenance contracts, specialized operator staffing, and facility requirements combine to create a high and relatively fixed operating cost base [8]. This cost profile does not reflect the actual processing volume the institution generates on any given day.

The economic consequence leads to a type of structural inefficiency. Capacity provisioned to handle peak-period processing surges, such as “year-end escrow disbursement cycles,” sits underutilized during the remaining months. The institution effectively subsidizes peak-capacity insurance through its baseline operating expenses. Philippon's analysis of the fintech opportunity identifies this kind of infrastructure inefficiency as one of the primary drivers of financial technology adoption, as newer architectures allow institutions to align cost more closely with actual service consumption [8]. For institutions experiencing portfolio growth or significant shifts in the product mix, such a situation can lead to a material financial constraint due to the inability to align infrastructure costs with processing demand [18].

8.2 Variable-Cost Infrastructure and Workload-Responsive Scaling

The distributed processing model underlying the event platform enables a different cost structure. Compute, storage, and network resources are provisioned on demand in response to observed workload metrics. The cost of processing a given volume of events is proportional to that volume, rather than being anchored to a peak-capacity ceiling [20]. Katari and Dinesh Kalla demonstrate in their study of cloud-based financial data lakes that techniques such as tiered storage, auto-scaling compute clusters, and workload-aware resource allocation can produce substantial cost reductions compared to fixed-capacity infrastructure [18].

Dimension	Legacy mainframe	Distributed event platform
Capacity model	Fixed peak provisioning. Capacity sized to handle maximum projected portfolio volume. Unused capacity cannot be released or reallocated.	Demand-responsive provisioning. Resources allocated in proportion to active workload. Capacity expands and contracts automatically.
Cost variability	High fixed baseline. Hardware, licensing, facility, and staffing costs remain largely constant regardless of processing volume. Off-peak periods generate no savings.	Variable with workload. Compute, storage, and network costs scale with event volume. Off-peak periods reduce expenditure proportionally.
Scaling mechanism	Vertical only. Capacity increases require hardware upgrades with procurement and installation lead times measured in weeks to months.	Horizontal and automated. Additional processing nodes provisioned in response to queue-depth metrics within minutes, without service interruption.
Peak provisioning requirement	Must sustain permanent overcapacity. Month-end and year-end surge capacity must be maintained year-round, creating sustained infrastructure subsidy for peak periods.	Surge capacity provisioned on demand. Processing nodes added for month-end cycles and released afterward. No permanent peak-capacity overhead.
Operational staffing profile	Specialized, high-cost roles. Mainframe operations require certified engineers for hardware management, batch scheduling, and system programming. Skills are scarce.	Commodity cloud operations skills. Distributed platform operations leverage broader talent pools. Infrastructure-as-code reduces manual operational dependency.
Portfolio growth absorption	Requires capital investment. Doubling portfolio size requires proportional hardware expansion with capital expenditure approval cycles.	Absorbed within variable cost structure. Growth was reflected in operating expenditure without discrete capital investment decisions.
Security and compliance controls	Controls embedded in physical infrastructure perimeter. Change management processes are well-established but slow.	Controls defined as code and applied consistently across dynamically provisioned resources per NIST SP 800-144 guidelines. Requires disciplined policy enforcement tooling.

Table 3: Comparison Between Legacy Mainframe vs. Distributed Event Platform [9, 19]

Variable-cost infrastructure requires careful architectural planning to function as intended. Resource provisioning must be fast enough to absorb intra-day workload spikes without creating a queue backlog that affects processing timelines. Deprovisioning must be governed by logic that prevents oscillatory scaling behavior in which resources are repeatedly acquired and released in response to short-duration fluctuations [4]. Security and data privacy controls, as outlined in NIST Special Publication 800-144, must be maintained consistently across dynamically provisioned resources to ensure that cloud-based scaling does not introduce compliance gaps [17].

8.3 Operational Transition Considerations

There is no flag-day cutover from a legacy mainframe-based mortgage servicing solution to a distributed event-orchestrated one. The complexity of mortgage accounting, the regulatory scrutiny over mortgage servicing operations, and the need for a guaranteed retention of historic loan data for reporting and analytical purposes demand a phased transition [1]. A more feasible transition architecture is to parallelize the event platform and the legacy system for an extended co-existence period. New originations can happen on the event platform while the legacy system is still running. The legacy loans are migrated in batches, or cohorts, by product type, regulatory complexity, and operational risk, with output constantly reconciled between the legacy and

new systems throughout the coexistence period to ensure that the new system is working correctly prior to the complete retirement of the old one [4]. This allows for rollback to legacy processing for subsets of cohorts that cannot use the new processing due to unforeseen issues that might occur, a risk control measure that is not atypical in a regulated financial environment. As Arner et al. explain, regulatory technology in financial institutions needs to be evidence-based, phased, and maintain regulatory compliance [7]. There is also the human aspect of this transition: employees with years of experience working with legacy systems must become acquainted with event-driven models, distributed monitoring tools, and workflows. According to the Mortgage Bankers Association, mortgage operations must factor in human and technology limitations when considering a platform of this magnitude [21].

9. Evaluation and Observations

The evaluation is grounded in prototype implementation observations and systematic comparative analysis conducted against legacy batch-processing systems. The prototype reflects realistic mortgage servicing workloads, including payment processing, escrow adjustments, and delinquency workflows. Evaluation was conducted across four principal system dimensions: processing latency, scalability under increasing workload, deterministic consistency under replay conditions, and operational efficiency in rule deployment.

- **Latency Reduction:** Comparative evaluation demonstrates a significant reduction in processing latency from batch-scale delays to near-real-time execution, eliminating the intra-day information gap that constrains downstream delinquency monitoring and customer inquiry systems [4, 20].
- **Scalability:** Partitioned event streams exhibit near-linear throughput scalability as additional processing nodes are introduced, consistent with the log-based partitioning model described by Kreps et al. [5] and the distributed transaction guarantees outlined by Thomson et al. [26]. This characteristic remained stable across workload ranges representative of institutional mortgage portfolio sizes, corroborating the theoretical scalability properties of the partition-by-loan-identifier strategy under production-scale conditions.
- **Deterministic Consistency:** Replay-based validation yielded no observed divergence in financial outputs across repeated executions under controlled conditions. Fixed-point arithmetic and canonical event-time processing, in alignment with IEEE 754-2019 [16] and the immutability principles established by Helland [24, 25], eliminated platform-dependent rounding discrepancies observed in legacy systems. The reproducibility of outputs across independent replay executions substantiates the event log as a formally reliable and auditable basis for financial computation.
- **Operational Efficiency:** Externalized rule management reduces deployment cycles from weeks to hours. Shadow processing validation enables regulatory rule changes to be evaluated against live event streams prior to production activation, substantially reducing deployment risk [3, 14]. The capacity to validate rule modifications against production-representative data without affecting posted ledger state constitutes a structural improvement over the exhaustive regression cycles necessitated by tightly coupled legacy architectures.

Collectively, these findings substantiate the architectural claims advanced in this paper. The observed improvements in latency, scalability, deterministic consistency, and operational efficiency are not incidental outcomes but direct consequences of the structural properties the proposed architecture was designed to provide. Taken as a whole, the evaluation supports the conclusion that event-driven processing represents a technically sound and operationally viable foundation for deterministic mortgage servicing at institutional scale.

Conclusion

In this paper, it is demonstrated that each of batch latency, code/data tight coupling, and capacity-constrained infrastructure in legacy mortgage-servicing systems is an architectural problem. Since these are architectural problems, there is also an architectural solution: an event-driven architecture. Our solution addresses the problem by treating the

event log as the canonical source of truth, hosting the services that process its events in a stateless manner, and externalizing decision management to eliminate ambiguity, enforce consistency, and reduce regulatory response time from weeks to hours. The layered idempotency model provides exactly-once ledger semantics that are unattainable in legacy systems, and the unified compliance audit trail delivers a single durable record that satisfies regulatory traceability requirements, both financial and procedural. Evaluation results demonstrate that these properties enable near-real-time processing, near-linear scalability, and deterministic replay under controlled conditions. By adopting this architecture, firms are not only upgrading their infrastructure but also building the ability to absorb regulatory change, scale as their portfolio grows, and provide the level of audit accountability that is increasingly required by regulators and counterparties.

References

[1] Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2012. Available: https://raw.githubusercontent.com/ZoranLi/Books1/master/Patterns%20of%20Enterprise%20Application%20Architecture.pdf

[2] Gregor Hohpe and Bobby Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Professional, 2004. Available: https://ptgmedia.pearsoncmg.com/images/9780321200686/samplepages/0321200683.pdf

[3] Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004. Available: https://msstest.sankuai.com/v1/mss_156341e12eb248878e532dd820706c40/mall-data-init/ddd-en.compressed.pdf

[4] Martin Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, O'Reilly Media, 2017. Available: https://books.google.com/books?id=p1heDgAAQBAJ

[5] Jay Kreps, Neha Narkhede, and Jun Rao, "Kafka: A Distributed Messaging System for Log Aggregation," in Proc. NetDB Workshop, Athens, Greece, 2011. Available: https://pages.cs.wisc.edu/~akella/CS744/F17/838-CloudPapers/Kafka.pdf

[6] Pat Helland, "Immutability Changes Everything," Communications of the ACM, 2015. Available: https://dl.acm.org/doi/pdf/10.1145/2844112

[7] Douglas W. Arner, Janos Barberis, and Ross P. Buckley, "FinTech, RegTech, and the Reconceptualization of Financial Regulation," Northwestern Journal of International Law and Business, 2016. Available: https://scholarlycommons.law.northwestern.edu/cgi/viewcontent.cgi?article=1817&context=njilb

[8] Thomas Philippon, "The FinTech Opportunity," NBER Working Paper No. w22476, National Bureau of Economic Research, 2016. Available: https://www.nber.org/system/files/working_papers/w22476/w22476.pdf

[9] Consumer Financial Protection Bureau, "12 CFR Part 1024 – Real Estate Settlement Procedures Act (Regulation X)," 2023. Available: https://www.consumerfinance.gov/rules-policy/regulations/1024/

[10] Paris Carbone et al., "Apache Flink: Stream and Batch Processing in a Single Engine," IEEE Bulletin of the Technical Committee on Data Engineering, 2015. Available: https://research.tudelft.nl/en/publications/apache-flink-stream-and-batch-processing-in-a-single-engine/

[11] Michael Armbrust et al., "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in Proc. 2018 International Conference on Management of Data, 2018. Available: https://dl.acm.org/doi/pdf/10.1145/3183713.3190664

[12] James C. Corbett et al., "Spanner: Google's Globally Distributed Database," ACM Transactions on Computer Systems, vol. 31, no. 3, pp. 1–22, 2013. Available: https://dl.acm.org/doi/pdf/10.1145/2491245

[13] Edward A. Lee et al., "Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems," Intelligent Computing, 2023. Available: https://spj.science.org/doi/pdf/10.34133/icomputing.0013

[14] Wenquan Jin et al., "Dynamic Inference Approach Based on Rules Engine in Intelligent Edge Computing for Building Environment Control," Sensors, 2021. Available: https://www.mdpi.com/1424-8220/21/2/630

[15] Ebimor Yinka Gbabo, Odira Kingsley Okenwa, and Possible Emeka Chima, "Modeling Audit Trail Management Systems for Real-Time Decision Support in Infrastructure Operations," Shodhshauryam, International Scientific Refereed Research Journal, 2023. Available: https://shisrrj.com/paper/SHISRRJ236153.pdf
[16] IEEE Standard Association, "IEEE Standard for Floating-Point Arithmetic – IEEE Std 754-2019," IEEE Computer Society, 2019. Available: https://www-users.cse.umn.edu/~vinals/tspot_files/phys4041/2020/IEEE%20Standard%20754-2019.pdf
[17] Wayne Jansen and Timothy Grance, "Guidelines on Security and Privacy in Public Cloud Computing," NIST Special Publication 800-144, National Institute of Standards and Technology, 2011. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-144.pdf
[18] Abhilash Katari and Dinesh Kalla, "Cost Optimization in Cloud-Based Financial Data Lakes: Techniques and Case Studies," ESP Journal of Engineering and Technology Advancements, vol. 1, no. 1, pp. 150–157, 2021. Available: https://doi.org/10.56472/25832646/JETA-V1I1P116
[19] Martin Fowler, "Event Sourcing," martinfowler.com, 2005. Available: https://martinfowler.com/eaaDev/EventSourcing.html
[20] Microsoft, "Event-Driven Architecture Style," Microsoft Azure Architecture Center, 2026. Available: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven
[21] Mortgage Bankers Association, "AI in Mortgage Industry," MBA Policy Report, 2024. Available: https://www.mba.org/docs/default-source/policy/27251-mba-policy-state-ai-report.pdf
[22] Consumer Financial Protection Bureau, "Consumer Financial Protection Bureau Publishes Assessments of Ability-to-Repay and Mortgage Servicing Rules," CFPB Newsroom, 2019. Available: https://www.consumerfinance.gov/about-us/newsroom/consumer-financial-protection-bureau-publishes-assessments-ability-repay-and-mortgage-servicing-rules/
[23] Consumer Financial Protection Bureau, "Mortgage Servicing Rules Under the Truth in Lending Act (Regulation Z)," CFPB Final Rules, 2021. Available: https://www.consumerfinance.gov/rules-policy/final-rules/mortgage-servicing-final-rules-mortgage-servicing-rules-under-truth-lending-act-regulation-z/
[24] Pat Helland, "Life beyond distributed transactions: an apostate's opinion," Queue, 2016. Available: https://spawn-queue.acm.org/doi/pdf/10.1145/3012426.3025012
[25] Pat Helland, "Immutability changes everything," Communications of the ACM, 2015. Available: https://dl.acm.org/doi/pdf/10.1145/2844112
[26] Alexander Thomson et al., "Calvin: fast distributed transactions for partitioned database systems," In Proceedings of the 2012 ACM SIGMOD international conference on management of data, 2012. Available: https://dl.acm.org/doi/pdf/10.1145/2213836.2213838