International Journal of Advanced Research in Engineering Technology & Science
ISSN: 2394-2819
Impact Factor: 7.10
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Email: editor@ijarets.org Volume-11, Issue-3 March – 2024 www.ijarets.org

A Reference Architecture for Migrating Consumer FinTech Monoliths to Event-First Microservices

Naresh Kumar Paturi
Principal Architect, Greenlight Financial Technology, Inc.

Partha Roy
Technical Lead, *Tata Consultancy Services*

Abstract. Consumer financial platforms usually begin as a single deployable application, because in the early years product velocity matters more than clean platform boundaries. As card processing, billing, and money movement mature, those same domains begin to need isolation, independent release, and reliability that can be measured rather than assumed. This paper presents a reference architecture and migration method for an event-first re-platforming, grounded in the first author's work on the Money, Billing, and Card Processing domains of a family-finance platform. The architecture decomposes the system along domain ownership seams, gives each extracted service its own write model, publishes state changes through a transactional outbox, and protects financial correctness with idempotency keys, deterministic state machines, and continuous reconciliation rather than with assumptions about perfect message delivery. A staged strangler procedure allows services to be extracted while the platform stays available and auditable. We define a measurement methodology using deployment frequency, change-failure rate, high-percentile latency, and service-ownership maturity, and we specify the evidence required to validate each reported outcome. The reported improvements, including a roughly fivefold increase in deployment frequency, an approximately thirty percent reduction in change-failure rate, and meaningful latency reductions on core paths, are drawn from this work and are presented as case-study figures that should be confirmed with continuous-integration and observability data before they are treated as fully generalizable results. The contribution is a repeatable architecture, a migration playbook, and an evidence model that other high-volume, regulated FinTech teams can adopt.

Index Terms. FinTech architecture, microservices, event-first design, domain-driven design, idempotency, transactional outbox, strangler migration, reliability.

I. Introduction

A family-finance platform cannot treat money movement as ordinary web traffic. When a parent loads a child's card before a school day, when an allowance transfers between accounts, or when an authorization is decided at a point of sale, a failure is not a cosmetic defect. A duplicated transfer, a dropped authorization, or an inconsistent ledger entry is experienced immediately by a customer as a loss of trust, and in a regulated environment it can also become a compliance event. Correctness is therefore not one quality attribute among many; it is the precondition for everything else the platform does.

Most such platforms start as a monolith for sound reasons. A single codebase and a single database keep early development fast, make local transactions simple, and avoid the operational overhead of a distributed

system before the product has found its shape. The same properties that help at the start become liabilities at scale. A shared deployment couples the release schedule of every team, so a change to a peripheral feature can block a critical fix. A shared database couples scaling limits and data models. A single process widens the blast radius of any fault, so an error in one domain can degrade the whole platform. As the Money, Billing, and Card Processing domains grow, these couplings begin to dominate both delivery speed and reliability.

The engineering problem is therefore narrower and harder than the familiar instruction to split a monolith. It is how to increase delivery independence and domain ownership while preserving financial correctness, and to do so on a system that must remain available and auditable throughout the migration. Payments work cannot pause operations, tolerate ambiguous ledger states, or accept a window in which transactions are unaccounted for. The migration must keep the books correct while the structure beneath them is rebuilt.

This paper turns the first author's work on this migration into a reference architecture with an explicit evidence model. The approach combines domain decomposition along ownership seams, idempotent service interfaces, outbox-driven event publication, backpressure and retry policy, and a staged strangler procedure with production-readiness gates. Its distinguishing claim is not the invention of any single pattern, since the constituent patterns are well established, but their consolidation under one governance model in which correctness, auditability, and release independence are treated as a single design objective.

The contributions of this paper are the following.

- A reference architecture for event-first decomposition of money, billing, and card domains that preserves financial correctness during and after migration.
- A migration method based on the strangler pattern, with idempotent interfaces, transactional-outbox event publication, and reconciliation as the safety net rather than broker guarantees.
- A set of production-readiness gates and migration playbooks that make the procedure repeatable across teams rather than dependent on individual expertise.
- A measurement methodology, with precise metric definitions and the evidence required to validate each claim, so that reported gains can be substantiated rather than asserted.

The remainder of the paper is organized as follows. Section II reviews related work. Section III states the system model and requirements. Section IV presents the reference architecture. Section V discusses implementation. Section VI defines the evaluation methodology and presents the case data from this work. Sections VII through X cover discussion, threats to validity, future work, and conclusions.

II. Background and Related Work

The decomposition strategy rests on domain-driven design, in which service boundaries follow bounded contexts rather than technical layers, so that a service owns a coherent slice of the business and its data [1], [2]. This is the foundation that allows money, billing, and card processing to evolve and deploy independently without sharing a model that no single team fully controls.

The migration tactics draw on the microservices literature and, in particular, on guidance for moving incrementally from a monolith rather than rewriting it [3], [4], [5]. The strangler pattern provides the central technique: route traffic for a capability through a facade, build a replacement behind it, and retire the legacy path only once the replacement has proven equivalent under real load [6]. The catalog of integration and messaging patterns informs how services communicate once separated [7], and resilience patterns such as timeouts, circuit breakers, and bulkheads inform how failures are contained [8].

For data movement during and after extraction, the transactional outbox pattern allows a service to publish domain events atomically with its local state change, closing the gap in which state and events could diverge [9]. Where a business operation spans services and cannot share a single transaction, saga-style coordination provides compensation-based recovery [10], [11]. The broader theory here is well established: distributed systems should be designed for at-least-once delivery and idempotent processing rather than for an illusion of distributed ACID, a point argued forcefully in the transaction-processing and data-systems literature [12], [13], [14]. Idempotency as an interface property, with client-supplied keys and deterministic replay, has become standard practice in payment APIs [15].

What this work adds to the above is consolidation rather than a new primitive. Published treatments tend to describe each pattern in isolation. A regulated payments platform needs them combined under one governance model, with correctness, auditability, and release independence pursued together, and with reconciliation treated as a first-class control rather than an afterthought. The architecture below is an account of how those pieces fit together, and the evidence model is an account of how we would demonstrate that the combination worked.

III. System Model and Requirements

We model the platform as a set of business domains, each owning entities and a ledger-relevant state, that together must keep an authoritative record of money. The Money domain owns balances and transfers; Billing owns recurring charges and entitlements; Card Processing owns authorization and the card lifecycle. Each domain emits and consumes events. External actors include card networks, banking partners, and the customer-facing applications. We assume an at-least-once message substrate, partial failure as the normal case, and the need to remain operable and auditable at all times.

The functional requirements are that the platform authorize and post transactions, move money between accounts, apply billing events, and maintain a ledger that external parties and auditors can reconcile against. The non-functional requirements are the more demanding part of the problem. Correctness requires that no transaction is duplicated or lost and that the ledger is internally consistent. Availability requires that core paths survive the failure of individual services and dependencies. Auditability requires an immutable, explainable history of every state change. Release independence requires that a team can deploy its domain without coordinating a platform-wide release. Table I summarizes these requirements and the architectural mechanism that addresses each.

Table I. Requirements and addressing mechanisms

Requirement	Type	Addressing mechanism
No duplicate or lost transactions	Correctness	Idempotency keys, deterministic state machines, reconciliation
Internally consistent ledger	Correctness	Single write owner per domain, outbox-published events, repayable read models
Survive dependency failure	Availability	Timeouts, bulkheads, backpressure, fallback policy
Explainable history	Auditability	Immutable status history, event log, file and delivery proofs
Independent deployment	Delivery	Domain seams, versioned interfaces, decoupled data ownership

IV. Reference Architecture

The architecture uses domain seams as the migration boundary and as the runtime boundary. Each extracted service owns its write model, exposes idempotent interfaces for externally visible actions, and publishes

domain events through a transactional outbox. Read models are rebuilt from events where that is practical, which decouples query paths from write paths. The system favors what we call exactly-once-enough behavior: rather than assuming the broker delivers each message exactly once, it makes repeated delivery safe through idempotency, makes state transitions deterministic, and uses reconciliation to detect and resolve any residual divergence. Figure 1 shows the overall flow.

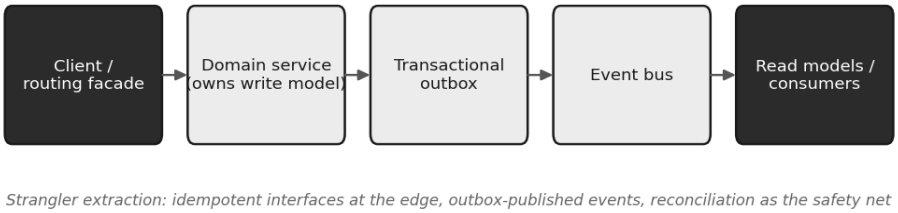

Fig. 1. Reference architecture and migration flow. Idempotent interfaces sit at the edge, services publish events through a transactional outbox, and reconciliation acts as the safety net behind eventual consistency.

A. Domain decomposition along ownership seams

Boundaries follow business ownership, not technical convenience. A seam is chosen where a domain has a coherent set of invariants and a team that can own them end to end. Money, Billing, and Card Processing each become an owned context with a single write authority for their data, which removes the shared-database coupling that made independent deployment impossible in the monolith.

B. Idempotent interfaces and exactly-once-enough semantics

Every externally visible action carries a client-supplied idempotency key. A repeated request with the same key returns the original outcome rather than performing the action again, which makes retries safe across network and broker failures [15]. Within a service, each money movement is a deterministic state machine, so that the same input always drives the same transition. Correctness does not depend on the broker delivering once; it depends on the action being safe to attempt more than once.

C. Outbox-driven event publication

A service records its state change and the event to be published in the same local transaction, writing the event to an outbox table; a separate process then relays outbox rows to the event bus [9]. This guarantees that an observed state change and its published event cannot diverge, which is the failure mode that produces the most damaging inconsistencies in a naive dual-write design. Change-data-capture is an alternative relay mechanism with the same guarantee [16].

D. Backpressure, retries, and failure isolation

Each dependency has an explicit timeout and a bounded retry budget, and independent workloads are isolated by bulkheads so that one slow dependency cannot exhaust shared resources [8]. Backpressure prevents a surge from cascading into a platform-wide failure. Retries are paired with idempotency so that a retried operation is safe by construction rather than by hope.

E. Migration playbooks

Extraction follows a staged strangler procedure. Traffic for a capability is routed through a facade; a new service is stood up behind it and run in shadow or dual mode against the legacy path; correctness is verified by reconciling the two; traffic is shifted incrementally; and the legacy path is retired only after equivalence is demonstrated under real load [6]. Each stage has a rollback. A service is not considered complete until it passes

production-readiness gates covering observability, alerting, runbooks, and incident response. Table II records the principal design decisions and the evidence that would justify each.

Table II. Design decision matrix

Design choice	Reliability	Latency	Evidence needed
Strangler extraction	High	Medium	Migration plan, traffic-split and shadow-run records
Transactional outbox	High	Medium	Outbox logs, relay lag, replay success rate
Idempotent interfaces	High	Low overhead	Idempotency-store metrics, duplicate-suppression rate
Synchronous service chain	Medium	Low initially	Timeout and cascade analysis
Shared database (transitional)	Low	Low	Deprecation plan, coupling map

V. Implementation Considerations

In the reference implementation the services are written in Kotlin and Java, deployed as containers on a managed cluster, and integrated through an event bus and managed queues. State is held in a combination of a relational store for strongly consistent ledger data and a key-value store for high-throughput access paths; single-table design and careful partition-key selection govern the latter [17], [18]. Infrastructure is provisioned as code, and delivery flows through an automated pipeline so that the deployment-frequency gains the architecture enables are actually realized in practice.

Observability is treated as part of the definition of done. Each service exports the indicators its owners are accountable for, emits structured events for audit, and ships with runbooks that define who acts during an incident and what may be retried safely. Production-readiness gates make these properties a precondition for taking traffic, which is what keeps the migration boring in production even as the structure changes underneath.

VI. Evaluation Methodology

A credible evaluation must compare a baseline monolith period against a post-migration period using objective signals, and it must normalize for activity so that improvements are not confused with reduced workload. We adopt four widely used delivery and reliability indicators, aligned with the research on software-delivery performance [19], [20]. Deployment frequency measures how often change reach production. Change-failure rate measures the fraction of changes that cause a degradation requiring remediation. High-percentile latency, reported at p95 and p99, measures user-facing responsiveness on core paths. Service-ownership maturity, assessed through an ownership rubric, captures whether teams genuinely operate their services.

The evidence required to validate each reported figure is specific. Deployment frequency and change-failure rate should come from continuous-integration and incident records over comparable windows. Latency should come from application performance monitoring on the same endpoints and cohorts before and after. Ownership maturity should come from a documented assessment rather than from impression. The figures presented here are reported and normalized values from this work, shown to illustrate the expected shape of the result; a final version should attach the exported pipeline and observability data, normalized by feature-release volume. Figure 2 shows the indexed case-study view, and Table III lists the reported figures alongside the artifact that would substantiate each.

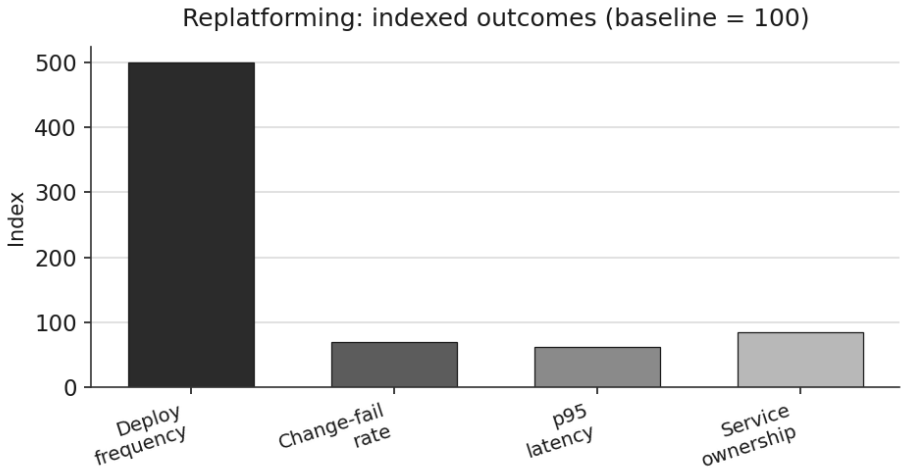

Fig. 2. Indexed case-study outcomes for the re-platforming.

Table III. Reported case metrics and required evidence

Metric	Reported / target value	Evidence to attach
Deployment frequency	~5x improvement (toward daily releases)	Pipeline export, release calendar
Change-failure rate	~30% reduction	Incident and change records
p95 latency (core paths)	30 to 40% reduction (path dependent)	APM dashboards, same cohorts
Core domains migrated	Money, Billing, Card Processing	Architecture diagrams, service inventory
Service-ownership maturity	Improved across teams	Ownership rubric assessment

VII. Discussion and Limitations

The lesson that matters most, in our experience, is that modernization succeeds when it is uneventful in production. The highest-value output of this work is often not a clever design but the rules, templates, and operating habits that a large group of engineers can follow safely under pressure. An architecture that many people can operate without re-deriving its invariants is worth more than an elegant one that only its designer can run.

Several limitations qualify the approach. The patterns generalize from high-volume, regulated payment systems and may be heavier than necessary in lower-volume or less-regulated settings. A partial migration leaves a transitional period in which old and new paths coexist, and the dual-write hazard during that period must be managed carefully, which is precisely why the outbox and reconciliation are emphasized. Finally, the reported gains reflect one organization's experience and should be read as indicative until reproduced with the evidence described above.

VIII. Threats to Validity

Internal validity depends on tying each reported figure to pipeline data, observability dashboards, incident records, or first-hand testimony before it is treated as evidence, and on normalizing by release volume so that a quieter period is not mistaken for a more reliable one. Construct validity depends on measuring the same endpoints and cohorts before and after, since latency and failure numbers are easy to bias through inconsistent

scoping. External validity is bounded to high-volume, regulated payments. On confidentiality, production details, customer identifiers, exact volumes, and partner terms are sanitized here unless disclosure is authorized. On reporting integrity, the figures are drawn from the first author's work on the platform, and we will attach sanitized case evidence to support each before treating any figure as a final empirical result.

IX. Future Work

Three directions would strengthen the result. The first is a controlled reproduction with exported delivery and reliability data and an explicit normalization for release volume. The second is a formal treatment of the reconciliation control, quantifying detection time and false-positive rate for divergences under varying late-arrival distributions. The third is an extension of the ownership-maturity rubric into a measurable construct that can be tracked over time, so that organizational maturity becomes a reported outcome rather than a qualitative claim.

X. Conclusion

This paper presented a reference architecture and migration method for moving consumer FinTech monoliths to an event-first microservices design without interrupting regulated financial workflows. The contribution is the consolidation of established patterns under one governance model, a repeatable strangler procedure with production-readiness gates, and an evidence model that ties each reported outcome to a verifiable artifact. The next step is to replace illustrative values with exported data and align the manuscript to the target venue.

Author contributions. The monolith-to-microservices migration described here was carried out by the first author (N. K. Paturi) at Greenlight Financial Technology, Inc. Both authors contributed to the reference architecture, the migration method, and the evaluation and evidence model presented in this paper.

References

[1] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003.
[2] V. Vernon, Implementing Domain-Driven Design. Addison-Wesley, 2013.
[3] S. Newman, Building Microservices, 2nd ed. O'Reilly, 2021.
[4] S. Newman, Monolith to Microservices. O'Reilly, 2019.
[5] C. Richardson, Microservices Patterns. Manning, 2018.
[6] M. Fowler, "Strangler Fig Application," martinfowler.com, 2004. https://martinfowler.com/bliki/StranglerFigApplication.html
[7] G. Hohpe and B. Woolf, Enterprise Integration Patterns. Addison-Wesley, 2003.
[8] M. T. Nygard, Release It!, 2nd ed. Pragmatic Bookshelf, 2018.
[9] Amazon Web Services, "Transactional Outbox Pattern," AWS Prescriptive Guidance. https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/transactional-outbox.html
[10] Microsoft Azure Architecture Center, "Saga distributed transactions pattern." https://learn.microsoft.com/en-us/azure/architecture/patterns/saga
[11] H. Garcia-Molina and K. Salem, "Sagas," in Proc. ACM SIGMOD Int. Conf. Management of Data, 1987, pp. 249–259.
[12] P. Helland, "Life beyond Distributed Transactions: an Apostate's Opinion," Commun. ACM, vol. 60, no. 2, 2017.
[13] M. Kleppmann, Designing Data-Intensive Applications. O'Reilly, 2017.
[14] P. A. Bernstein and E. Newcomer, Principals of Transaction Processing, 2nd ed. Morgan Kaufmann, 2009.
[15] Stripe, "Designing robust and predictable APIs with idempotency," Stripe Blog, 2017. https://stripe.com/blog/idempotency
[16] Debezium, "Change Data Capture," Debezium Documentation, 2022. https://debezium.io/documentation/
[17] Amazon Web Services, "Best practices for designing and using partition keys in DynamoDB." https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-table-design.html
[18] A. DeBrie, The DynamoDB Book. 2020.
[19] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. IT Revolution, 2018.