IJAREEIE

e-ISSN: 2278-8875
p-ISSN: 2320-3765

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering

Volume 12, Issue 3, March 2023

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

Impact Factor: 8.317

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering (IJAREEIE)

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

Migrating Monolithic Banking Applications to Microservices: An Architectural Case Study in ACH and Wire Payment Processing

Dinesh Nallapareddy

Sr Full Stack Developer, USA

ABSTRACT: This case study documents the architectural decomposition of a core banking monolith's payment functions into independently deployable microservices, focused on two payment rails with sharply different operating characteristics: batch-oriented ACH transfers and real-time, irrevocable wire transfers. It works through the strangler fig migration strategy, the resulting bounded contexts for ACH processing, wire processing, sanctions screening, and ledger settlement, and the tactical patterns, including change data capture, the outbox pattern, idempotency keys, and three-way reconciliation, that kept the system correct during an extended dual-run period. Illustrative figures and tables throughout show reference architectures, message flows, migration timelines, and representative before-and-after performance and reliability metrics. The material closes with regulatory considerations specific to payment processing, a worked cutover plan, and a catalog of anti-patterns observed during comparable modernization programs.

KEYWORDS: ACH, Fedwire, SWIFT, NACHA, monolith decomposition, strangler fig, microservices, dual-run migration, outbox pattern, idempotency, payment reconciliation, circuit breaker, BSA/AML, OFAC screening, core banking modernization

I. INTRODUCTION

Core banking platforms built in the 1990s and 2000s were designed around a single relational database and a single deployable application, an architecture that matched the batch-oriented, overnight processing model of that era's payment rails. Automated Clearing House, or ACH, transfers were designed from the outset around batch files and settlement windows, so a monolithic, batch-scheduled application was a reasonable fit. Wire transfers, in contrast, have always settled individually and irrevocably within seconds, and forcing wire processing through the same batch-oriented codebase as ACH processing has, over time, become a source of both operational risk and missed opportunity as customer expectations shifted toward real-time payment visibility.

This case study follows a representative modernization program that decomposed a core banking monolith's payment functions into a set of independently deployable microservices, organized around the distinct operating models of ACH and wire processing. The program is presented as a composite, illustrative case rather than a single named institution's deployment, so that the architectural reasoning can be examined without being tied to any particular vendor platform or proprietary detail.

Table.1. Scope of this case study

Scope area	Covered in this case study
Payment rails	ACH (batch, same-day, and standard) and Fedwire or SWIFT-based wire transfers
Migration approach	Strangler fig decomposition with an extended dual-run and reconciliation period
Target architecture	Independently deployable services integrated through domain events
Regulatory context	BSA/AML, OFAC sanctions screening, and core payment system risk management
Not covered	Card payments, real-time payment rail specifics beyond ACH and wire, retail lending

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

Sections 2 through 5 describe the starting state and the strategic decisions behind the migration. Sections 6 through 13 work through the tactical design of the ACH and wire services and the integration patterns connecting them. Sections 14 through 19 cover the operational and regulatory concerns specific to payment processing, and Sections 20 through 22 close with lessons learned, cost impact, and a summary.

II. THE LEGACY MONOLITH: ARCHITECTURE AND CONSTRAINTS

The starting architecture is a single Java application, deployed as one unit, backed by one relational database schema shared across every banking function. ACH and wire processing exist as modules within that application rather than as separately deployable services, which means a code change to wire transfer limits requires the same release process, the same regression test suite, and the same deployment window as a change to ACH batch formatting.

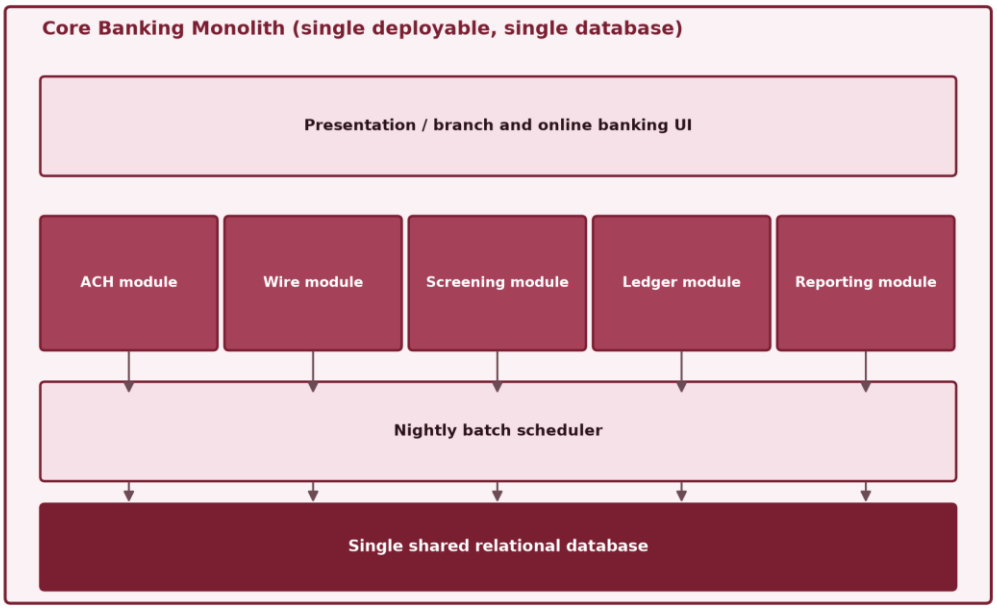


Figure.1. The legacy core banking monolith, showing shared modules over a single database.

Table.2. Monolith modules and their coupling characteristics

Module	Primary responsibility	Coupling to other modules
ACH module	NACHA file generation, batch scheduling, returns handling	Shares customer and account tables directly
Wire module	Fedwire and SWIFT message formatting and submission	Shares customer and account tables directly
Screening module	OFAC and sanctions list screening for both rails	Called synchronously in-process by ACH and wire code
Ledger module	Debit and credit posting, balance maintenance	Read and written directly by every other module
Reporting module	Regulatory and management reporting	Reads directly from live transactional tables

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

Table.3. Constraints imposed by the monolithic architecture

Constraint	Operational consequence
Single release train	A wire-specific fix waits for the next full regression cycle, typically two weeks
Shared database schema	A schema change for one module risks locking rows another module depends on
Nightly batch dependency	Same-day ACH windows compete with batch jobs for the same database capacity
In-process screening calls	A slow screening vendor call blocks the thread handling an unrelated wire request
Monolithic test suite	Full regression run exceeds six hours, discouraging frequent, small releases

None of these constraints were the result of poor engineering at the time the monolith was built; a single shared database was, for its era, the simplest way to guarantee that a debit and its matching credit stayed consistent. The constraints became liabilities only once the surrounding expectations, same-day settlement windows, continuous delivery, and service-specific scaling, moved faster than a single deployable unit could reasonably absorb.

III. DRIVERS FOR MIGRATION

Three categories of pressure converged on the decision to migrate: regulatory expectations that increasingly assume near-real-time visibility into payment status, operational risk concentrated in a single deployable unit, and a competitive need to support new payment experiences without a two-week release cycle.

Table.4. Selected regulatory drivers behind the migration

Regulatory driver	Detail
Same-day ACH expansion	Federal Reserve Bank Services same-day ACH windows require faster intraday processing
Sanctions screening currency	OFAC guidance expects screening against the most current list at the time of transfer
Operational resilience expectations	FFIEC business continuity guidance for critical payment systems

Table.5. Selected business and operational drivers behind the migration

Business or operational driver	Detail
Time to market	New payment features required a full monolith release even for wire-only changes
Incident blast radius	A single deployable meant a wire-processing defect could affect ACH availability
Talent and hiring	Newer engineers were harder to onboard onto a large, tightly coupled codebase
Infrastructure cost	The monolith was scaled as a single unit even though ACH and wire load profiles differ sharply

IV. MIGRATION STRATEGY: STRANGLER FIG AND INCREMENTAL DECOMPOSITION

The program adopted the strangler fig approach, a term coined by Fowler and developed further in the microservices literature by Newman (2015), in which new functionality is built as an independent service and a routing layer progressively redirects traffic to it, until the corresponding code in the monolith can be safely retired. The approach was chosen specifically because a full rewrite carried out in a regulated payments environment could not tolerate a long, high-risk cutover with no incremental validation.

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

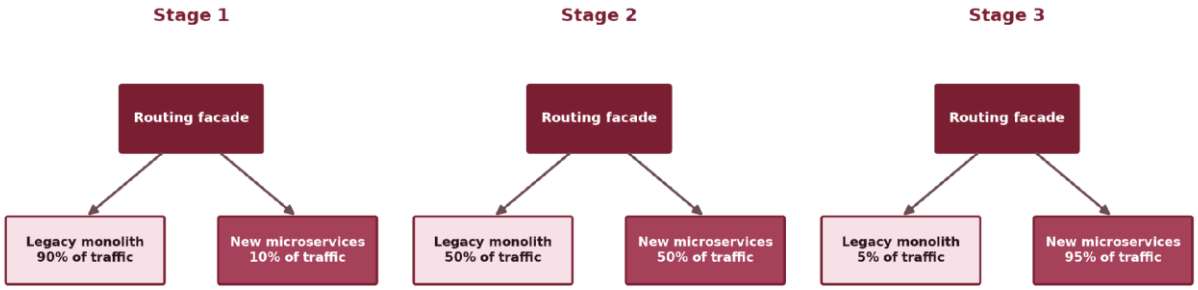


Figure.2. Strangler fig decomposition across three illustrative traffic-routing stages

Table.6. Migration approach options considered.

Migration approach	Risk profile	Suitability for this program
Full rewrite and single cutover	High; all risk concentrated at one event	Rejected; incompatible with regulatory risk tolerance
Strangler fig, incremental routing	Low per increment; validated continuously	Selected
Big-bang lift and shift to cloud infrastructure	Moderate; preserves coupling problems	Rejected; would not address the underlying coupling

Table.7. Illustrative strangler fig stages for the ACH service migration

Stage	Legacy share	New service share	Primary activity
Stage 1	90%	10%	Route a single low-risk account cohort to the new ACH service
Stage 2	50%	50%	Expand routing rules by account range; monitor reconciliation breaks
Stage 3	5%	95%	Route substantially all traffic; retain legacy for exception handling only

V. DOMAIN DECOMPOSITION: ACH AND WIRE PROCESSING BOUNDARIES

Strategic design, in the sense used by Evans (2003) and applied to microservices boundaries by Richardson (2018), asks where one problem ends and another begins before any code is written. ACH processing and wire processing look similar from a distance, both move money between accounts, but their operating models, failure modes, and regulatory obligations differ enough to justify separate bounded contexts rather than a single generalized payments service.

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

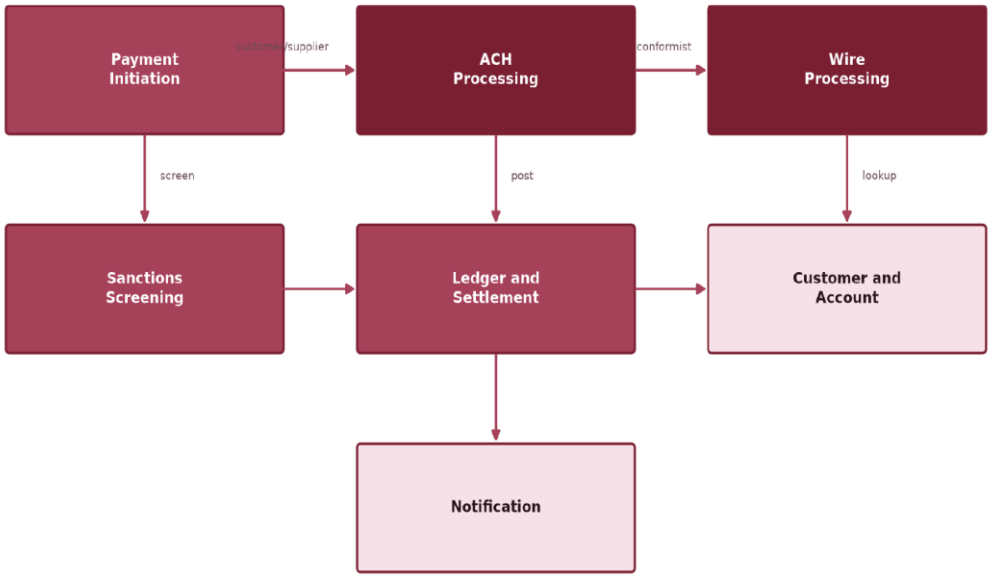


Figure.3. Bounded context map for the target payments architecture

Table.8. Domain classification for the target payments architecture, following Evans (2003)

Context	Classification	Rationale
Payment initiation	Core	Customer-facing entry point; differentiates the product experience
ACH processing	Core	Batch timing and returns handling are specific, high-value logic
Wire processing	Core	Irrevocable, real-time settlement logic with distinct risk controls
Sanctions screening	Core	Directly determines regulatory exposure; not safely outsourced to a generic tool
Ledger and settlement	Supporting	Necessary but follows well-understood double-entry accounting patterns
Customer and account	Generic	Reference data consumed by every context, owned upstream
Notification	Generic	Well-understood messaging problem

Table.9. Context mapping relationships across the target architecture

Relationship	Pattern	Notes
Payment initiation to ACH/Wire	Customer/supplier	Initiation negotiates its needs with each processing context
ACH/Wire to Card Issuance-style network rules	Conformist	Both rails adapt to externally defined network rules
ACH/Wire to Ledger	Published language	Ledger posting event schema is the stable integration contract
Processing contexts to Customer/Account	Shared kernel (identifiers only)	Only the account identifier scheme is shared, nothing else

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

VI. ACH PROCESSING: THE NACHA FILE LIFECYCLE AND BATCH WINDOWS

ACH transfers move through a well-defined lifecycle governed by the Nacha Operating Rules (Nacha, 2021): an originating company submits a file to its bank, the originating bank batches it for submission to an ACH operator, the operator sorts and forwards entries to receiving banks, and each receiving bank posts to the beneficiary account, all according to a small number of processing windows each business day.

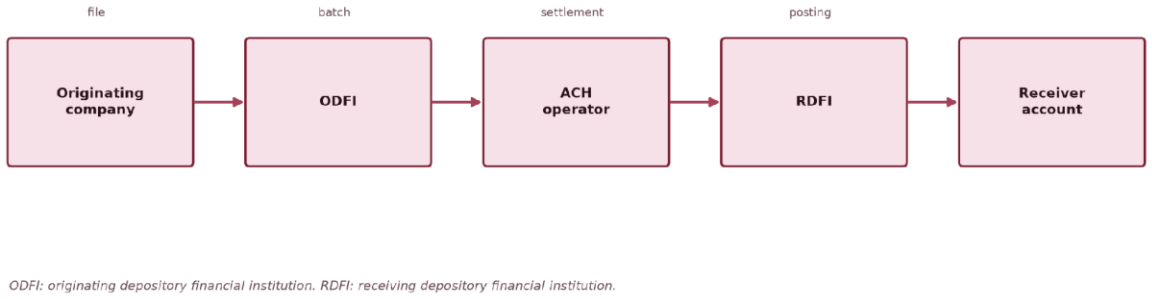


Figure.4. The NACHA file lifecycle from originating company to receiver account.

Table.10. Simplified structure of a NACHA-formatted ACH file

Field group	Example fields	Purpose
File header	Immediate origin, immediate destination, file creation date	Identifies the file and its routing
Batch header	Company name, standard entry class code, effective entry date	Groups entries with shared settlement timing
Entry detail	Transaction code, receiving DFI identification, amount	One record per individual transfer
Addenda (optional)	Payment-related information	Carries remittance detail for the receiver
Batch and file control	Entry counts, total debit and credit amounts	Integrity check against the detail records

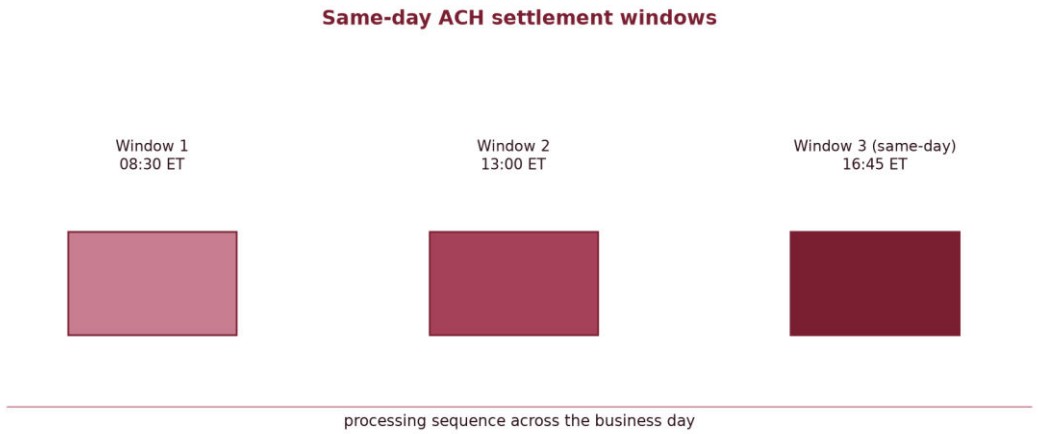


Figure.5. Same-day ACH settlement windows across a business day

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

Table.11. Selected NACHA standard entry class codes relevant to the migrated service

Standard entry class code	Typical use
PPD	Prearranged payment and deposit, consumer accounts
CCD	Corporate credit or debit, business-to-business
WEB	Internet-initiated consumer entries
TEL	Telephone-initiated consumer entries

VII. WIRE PROCESSING: FEDWIRE AND SWIFT MESSAGE FLOWS

A wire transfer follows a fundamentally different model from ACH: there is no batch, no settlement window, and no netting. Each transfer is submitted, transmitted, and settled individually, typically within seconds for a domestic Fedwire transfer, and the settlement is final and irrevocable once confirmed by the network (Federal Reserve Financial Services, 2021).

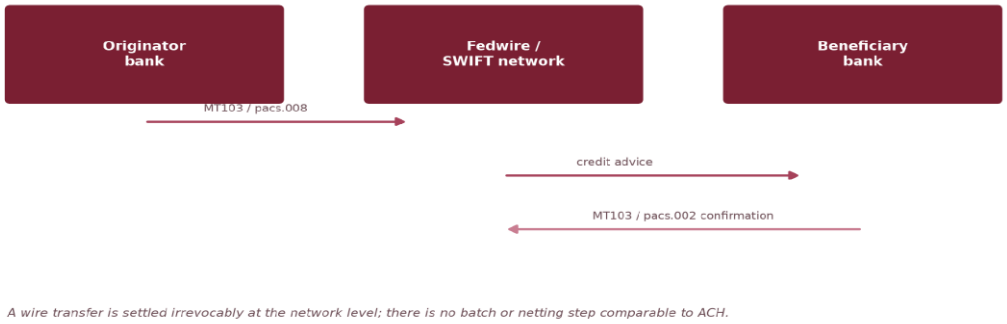


Figure.6. A representative wire transfer message flow across the originator bank, the network, and the beneficiary bank.

Table.12. ACH and wire compared on properties relevant to service design.

Property	ACH	Wire
Settlement timing	Batched, within defined windows	Individual, near-real-time
Revocability	Returnable within defined return windows	Irrevocable once settled
Typical message standard	NACHA file format	Fedwire format, or SWIFT MT103 / ISO 20022 pacs.008 internationally
Typical use case	Payroll, recurring bills, vendor payments	Real estate closings, large or time-critical transfers

Table.13. Selected wire message types relevant to the migrated service.

Message type	Direction	Purpose
MT103 / pacs.008	Originator bank to network	Customer credit transfer instruction
MT202 / pacs.009	Bank to bank	Financial institution transfer, often covering an MT103
MT103 / pacs.002 confirmation	Network to originator bank	Settlement confirmation or rejection

Because a wire transfer cannot be recalled once settled, the wire service's design places its heaviest validation and screening logic before submission rather than relying on a post-hoc correction path the way the ACH service can rely on the return process described in Section 6. This asymmetry, front-loaded control for wire, back-loaded correction for

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

ACH, is one of the clearest illustrations of why the two rails were modeled as separate bounded contexts rather than as variants of one generic payment service.

VIII. TARGET MICROSERVICES ARCHITECTURE

The target architecture separates payment initiation, ACH processing, wire processing, sanctions screening, ledger and settlement, reconciliation, and notification into independently deployable services, integrated primarily through an append-only event bus rather than direct service-to-service calls, following the choreography style discussed in Hohpe and Woolf (2003) and Richardson (2018).

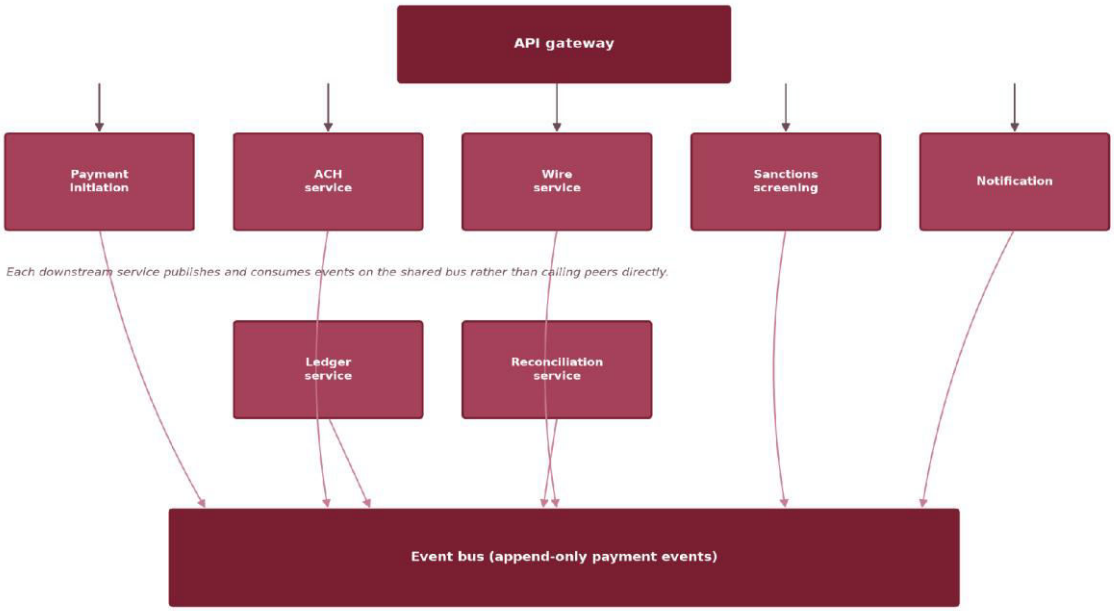


Figure.7. The target microservices architecture for payment processing

Table.14. Target service ownership and principal published events.

Service	Owns	Publishes
Payment initiation	Applicant-facing request validation	PaymentRequested
ACH service	NACHA file lifecycle and batch state	AchFileSubmitted, AchSettled, AchReturned
Wire service	Fedwire/SWIFT message lifecycle	WireSubmitted, WireSettled, WireRejected
Sanctions screening	Screening decisions and case history	ScreeningCleared, ScreeningHeld
Ledger service	Account balances and posting	EntryPosted
Reconciliation service	Cross-ledger break detection	ReconciliationBreakRaised

Table.15. Technology layers in the target architecture, described generically.

Layer	Representative technology category
API gateway	Managed API gateway or reverse proxy with authentication and rate limiting
Services	Independently deployed processes, one per bounded context
Event bus	Durable, ordered log-based messaging platform
Service data stores	One schema per service; no cross-service direct access
Observability	Distributed tracing, structured logs, metrics per service

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

IX. DATA MIGRATION AND DUAL-RUN STRATEGY

New services cannot simply start from an empty database; they must be populated from, and kept consistent with, the monolith's data for as long as the monolith remains the system of record. Change data capture, reading the monolith database's transaction log and republishing changes as events, is the mechanism used to keep new service data stores current without requiring the monolith's own code to change.

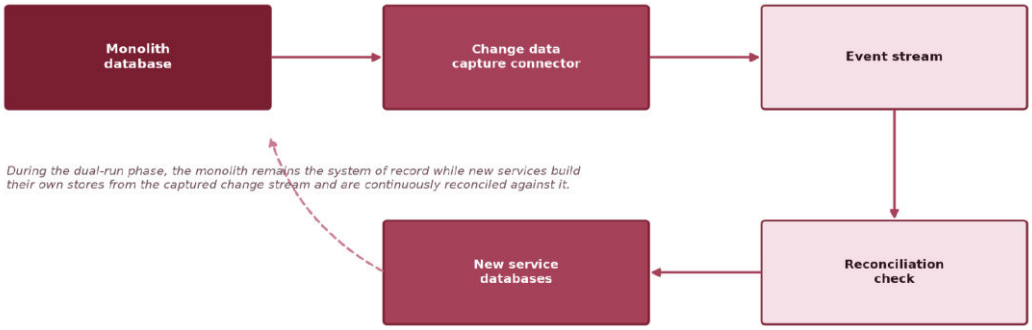


Figure.8. Change data capture feeding new service databases during the dual-run phase.

Table.16. Data ownership by migration phase

Phase	Monolith role	New service role
Dual-run	System of record; all writes originate here	Read replica built from captured changes; reconciled continuously
Shadow traffic	Continues processing; results compared, not acted upon	Processes a shadow copy of traffic for validation only
Live cutover	Read-only for migrated accounts	System of record for migrated accounts
Decommission	Retired for the migrated scope	Sole system of record

Table.17. Data consistency risks during the dual-run phase and their mitigations.

Consistency risk	Mitigation
Change data capture lag	Continuous reconciliation with an alert threshold on lag duration
Schema drift between monolith and new store	Versioned transformation layer at the capture boundary
Silent data loss during transition	Row-count and checksum reconciliation run on a fixed schedule

Change data capture lag deserves particular attention because it is the one risk in Table 17 that degrades gradually rather than failing outright, which makes it easy to miss without a dedicated metric. A capture connector that falls behind the monolith's transaction log does not raise an error; it simply means the new service's view of the world is a few seconds, then a few minutes, then eventually hours, out of date, while every other signal in the system continues to look healthy. The program treated lag as a first-class service level indicator from the first week of dual-run rather than adding it reactively after an incident, specifically because a slow, silent divergence between the monolith and the new services is far harder to diagnose after the fact than a hard failure would have been.

Reconciliation, covered in depth in Section 13, is the backstop for exactly this failure mode, but backstops are, by design, slower to react than a direct metric on the thing they are backstopping. Waiting for a reconciliation break to reveal capture lag, rather than alerting on lag directly, meant discovering the problem only after incorrect data had

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

already been read by a downstream consumer, which is a strictly worse outcome than catching the lag before any consumer acted on stale data.

X. EVENT-DRIVEN INTEGRATION AND THE OUTBOX PATTERN

Once services own their own data, a payment write and the event announcing that write must be consistent with each other. Writing to a database and publishing to a message broker as two separate operations risks one succeeding without the other. The outbox pattern avoids a distributed transaction by writing the business record and the outbound event to the same local database in a single transaction, then relaying the event asynchronously.

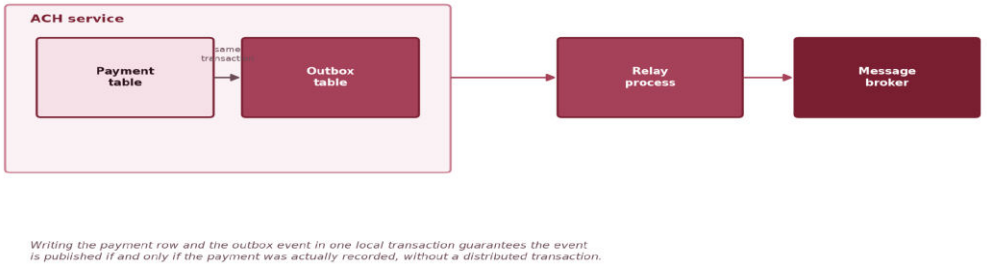


Figure.9. The outbox pattern applied to the ACH service.

Table.18. Approaches to consistent write-and-publish, compared

Approach	Consistency guarantee	Operational complexity
Distributed transaction (two-phase commit)	Strong, but coordinator becomes a single point of failure	High; poorly supported by modern brokers
Outbox pattern with relay	Strong for the local write; at-least-once delivery downstream	Moderate; requires a relay process and duplicate handling
Best-effort dual write, no outbox	Weak; a crash between the two writes loses consistency	Low, but unsafe for payment data

XI. API GATEWAY AND LEGACY FACADE DESIGN

The routing facade introduced in Section 4 is the component that makes the strangler fig migration possible in practice. It sits in front of both the monolith and the new services and decides, per request, which one should handle it.

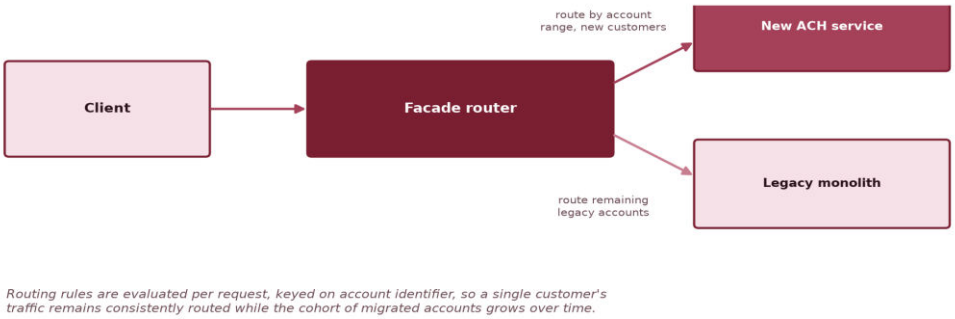


Figure.10. The facade router directing traffic between the legacy monolith and the new ACH service

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

Table.19. Routing key options for the legacy facade, compared.

Routing key	Suitability	Notes
Account identifier	High	Stable per customer; avoids splitting one customer's traffic inconsistently
Request timestamp	Low	Does not guarantee a customer's traffic stays consistently routed
Feature flag by percentage	Medium	Useful for early canary stages before account-based rules are ready

Table.20. Responsibilities of the legacy facade beyond simple routing

Facade responsibility	Detail
Request routing	Direct each request to the monolith or the appropriate new service
Response shape normalization	Present a stable API contract regardless of which backend served the request
Traffic shadowing	Optionally duplicate requests to the non-serving backend for comparison
Metrics and routing audit	Record which backend served every request, for reconciliation and rollback analysis

XII. IDEMPOTENCY AND EXACTLY-ONCE PAYMENT PROCESSING

A payment instruction must never be executed twice because a client retried a timed-out request. Idempotency keys, generated by the client and persisted by the service on first use, let a retried request return the original result rather than reprocess the payment, a pattern especially critical in payment processing where the cost of a duplicate is a real, incorrect movement of money.

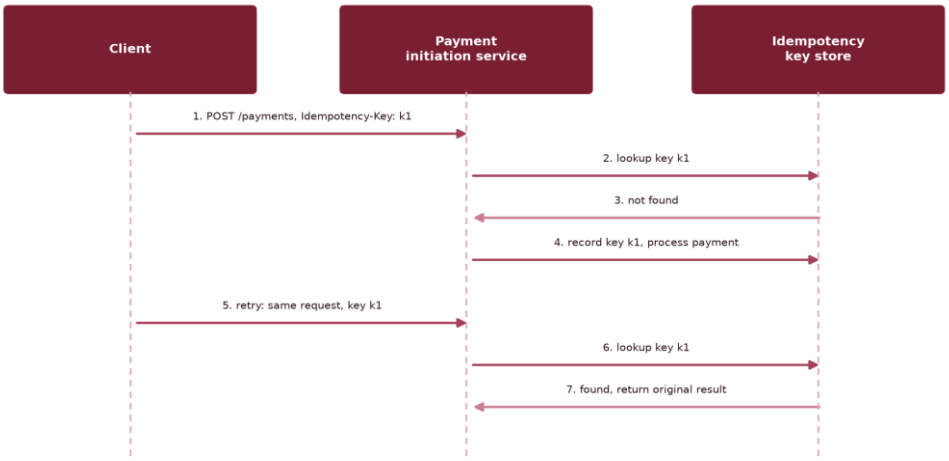


Figure.11. Idempotency key handling across an initial request and a client retry

Table.21. Idempotency key behavior across representative retry scenarios

Scenario	Without idempotency key	With idempotency key
Client retries after a timeout	Payment may be submitted twice	Second request returns the original result
Client retries after a network error with no response received	Payment may be submitted twice	Second request returns the original result
Two genuinely distinct payments, same amount	Both processed normally	Both processed normally, since keys differ

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

Table.22. Idempotency key design choices adopted for the payment initiation service.

Design choice	Recommendation
Key generation	Client-generated, unique per logical payment instruction
Key storage duration	Retained at least as long as the client's plausible retry window
Key scope	Scoped per endpoint and per client, not global

XIII. RECONCILIATION AND LEDGER CONSISTENCY

With ACH, wire, and ledger posting now handled by separate services rather than one transactional database, a three-way reconciliation between the core ledger and each payment rail's own sub-ledger became the primary control for detecting any divergence introduced by the new, more loosely coupled architecture.

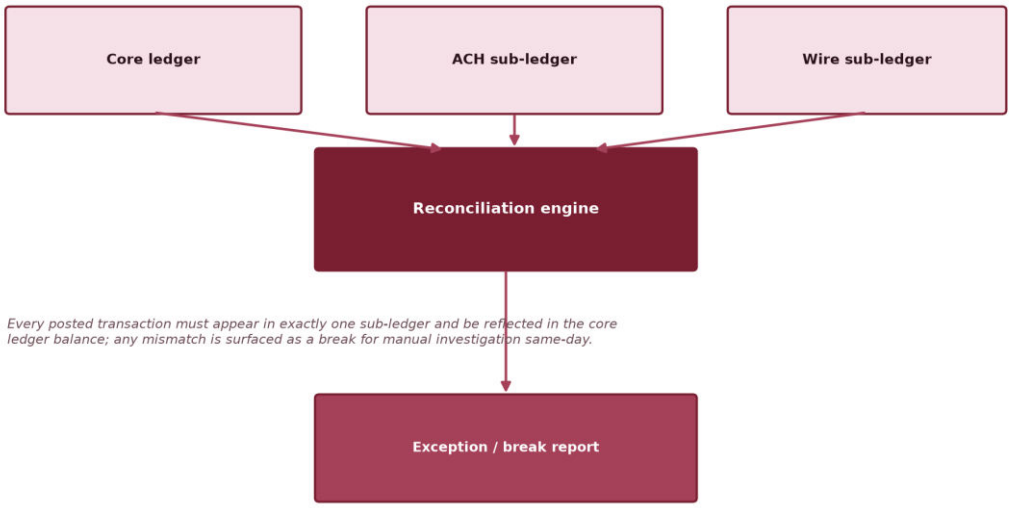


Figure.12. Three-way reconciliation across the core ledger and the ACH and wire sub-ledgers

Table.23. Reconciliation break types and resolution paths.

Break type	Typical cause	Resolution path
Entry in sub-ledger, missing from core ledger	Delayed or failed event delivery	Replay the missing posting event
Entry in core ledger, missing from sub-ledger	Sub-ledger write failure after posting succeeded	Reprocess from the source event log
Amount mismatch between ledgers	Rounding or currency conversion discrepancy	Manual investigation and correcting entry

Table.24. Illustrative reconciliation service level targets

Reconciliation SLA	Target
Same-day break detection	Within four hours of the affected settlement window
Break resolution for amounts under a defined threshold	Same business day
Break resolution for amounts above the threshold	Escalated to manual review within one hour of detection

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

XIV. SECURITY AND REGULATORY COMPLIANCE

Decomposing the monolith does not reduce its regulatory obligations, and in several respects it increases the surface area that must demonstrate compliance, since each new service boundary is also a boundary the institution must show is correctly controlled.

Table.25. Bank Secrecy Act and anti-money-laundering controls preserved during migration

BSA/AML control	How it is preserved in the target architecture
Suspicious activity monitoring	Reconciliation and screening events feed a dedicated monitoring pipeline
Recordkeeping	Every service's outbox and event log provides an immutable audit trail
Currency transaction reporting thresholds	Enforced in the payment initiation service, upstream of both rails

Table.26. OFAC sanctions screening considerations in the target architecture

OFAC screening concern	Design response
Screening currency (up-to-date list)	Sanctions screening service polls list updates independently of release cycles
Screening completeness across both rails	Both ACH and wire services call the same screening service, no bypass path
False positive handling	Screening holds route to a case queue rather than blocking silently

Consolidating screening behind a single service also simplified an examination process that had, under the monolith, required auditors to trace two separate in-process call paths, one embedded in the ACH module and one embedded in the wire module, to confirm both actually invoked the same screening logic against the same list version. With a single shared screening service, that confirmation reduces to verifying one dependency rather than reconciling two independent implementations that happened to agree in practice but were never guaranteed to by the code itself.

Table.27. Data classification applied consistently across the new services.

Data classification	Handling rule
Payment instruction data	Encrypted at rest and in transit; access scoped to the owning service
Screening case data	Retained per regulatory record retention schedules
Reconciliation break data	Access restricted to operations and compliance roles

XV. RESILIENCY FOR PAYMENT RAILS

Wire processing in particular depends on a small number of external network connections, and a resilient design assumes any one of them can degrade. The wire service applies the circuit breaker pattern (Fowler, 2014; Nygard, 2018) to its primary network connection and fails over to a backup path when the primary's failure rate crosses a threshold.

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

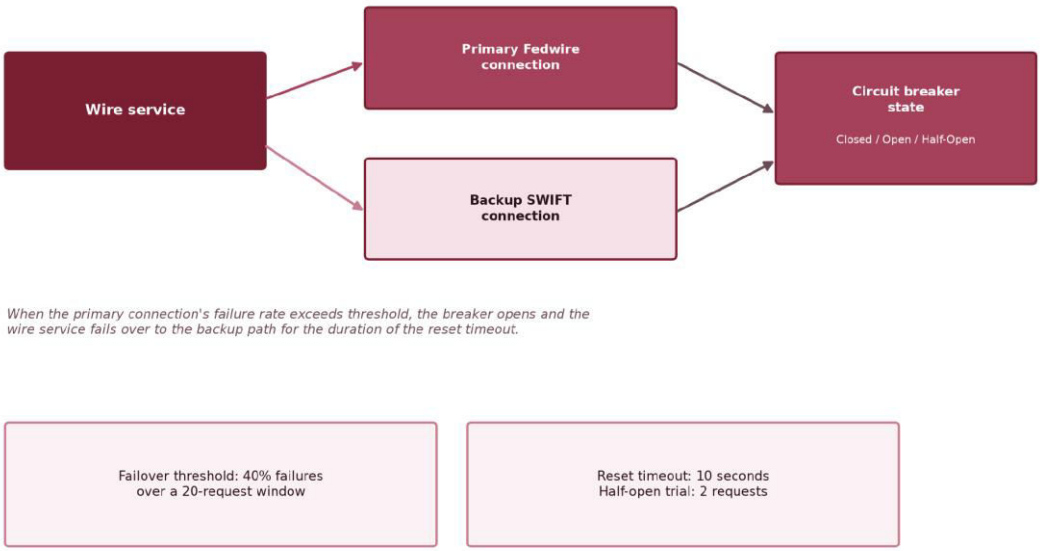


Figure.13. Circuit breaker and failover between primary and backup wire network connections.

Table.28. Failure modes on the wire processing path and the corresponding resiliency response

Failure mode	Detection signal	Response
Primary network connection degraded	Rising failure rate over a rolling window	Circuit opens; traffic fails over to backup path
Backup path also degraded	Failure rate rises on backup after failover	Escalate to manual intervention; queue for retry
Screening service slow	Timeout on screening calls	Bulkhead isolates screening calls from ledger and network calls

Table.29. Resiliency patterns applied across the wire and ACH services

Resiliency pattern	Applied to
Circuit breaker	Fedwire and SWIFT network connections
Bulkhead isolation	Sanctions screening, ledger posting, and network submission call paths
Timeout with bounded retry	All outbound calls to network and screening dependencies

A failover drill, run on a recurring schedule rather than only during initial rollout, is what kept confidence in the backup SWIFT path honest over time. Network paths that are configured but never actually exercised have a well-documented tendency to have quietly drifted out of a usable state by the time they are needed, whether through an expired credential, a changed firewall rule, or a configuration change on one side that was never mirrored on the other. Scheduling a real failover, during a low-volume maintenance window rather than only simulating one, surfaced exactly this kind of drift twice during the program before it could have caused an actual outage.

XVI. OBSERVABILITY AND PAYMENT TRACING

A payment that once lived inside a single monolithic transaction now crosses five or six independently deployed services before it reaches a terminal state. A correlation identifier propagated across every hop, combined with per-service structured logging, is what lets an operator answer a specific customer inquiry about one payment's status without querying every service by hand.

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

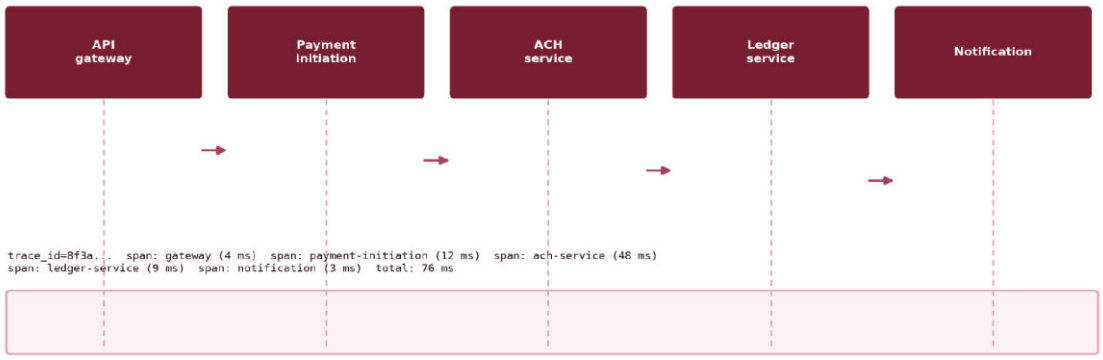


Figure.14. A payment trace spanning the gateway, initiation, ACH, ledger, and notification services

Metric	Legacy monolith	Target architecture
Visibility granularity	Single application log, mixed concerns	Per-service structured logs correlated by trace identifier
Time to answer a payment status inquiry	Manual database query by an operator	Trace lookup by correlation identifier, self-service
Alerting granularity	Application-wide alerts	Per-service alerts scoped to each service's own SLO

Table.30. Observability characteristics before and after decomposition

Service level objective	Target
ACH file submission latency	Within 5 minutes of the applicable cutoff time, 99.9 percent of files
Wire submission latency (p99)	Under 250 milliseconds from request to network submission
Trace availability	100 percent of payments traceable end to end within 1 minute of completion

Table 31. Illustrative service level objectives for the target payment services.

Correlation identifiers are only useful if every service actually propagates them, which in practice required a small platform library adopted uniformly across every service rather than a convention left to each team to implement independently. The one instance where a service initially generated its own identifier rather than propagating the inbound one produced a visible gap in the trace for every payment that passed through it, and was treated as a release-blocking defect once discovered, not a minor polish item, precisely because it broke the single capability, end-to-end traceability, that this section exists to provide.

XVII. CUTOVER STRATEGY AND ROLLBACK PLANNING

Cutover was executed as a progressive shift of transaction volume rather than a single event, with explicit rollback criteria defined before each stage began rather than improvised under pressure.

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

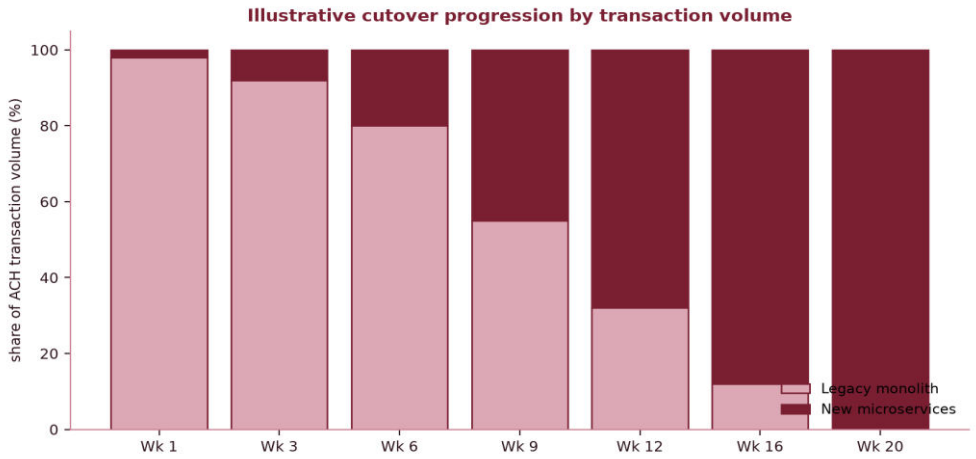


Figure.15. Illustrative cutover progression by ACH transaction volume across a twenty-week program

Table.32. Cutover stages and their explicit rollback triggers.

Cutover stage	Volume shifted	Rollback trigger
Pilot cohort	Under 5%	Any reconciliation break above a defined dollar threshold
Expansion	5% to 50%	Sustained increase in reconciliation break rate over three consecutive days
Majority cutover	50% to 95%	Any regulatory reporting discrepancy
Full cutover	95% to 100%	Confirmed data integrity issue affecting completed payments

Table.33. Rollback mechanisms available at each cutover stage

Rollback mechanism	Recovery time
Facade routing rule reversion	Minutes; no data migration required
Replay from event log for affected accounts	Hours, depending on volume affected
Full program pause pending investigation	Indefinite, governed by incident response process

XVIII. PERFORMANCE AND CAPACITY BENCHMARKING

Benchmarking compared the legacy monolith and the target architecture on measures specific to each rail's operating model: ACH batch processing time, since ACH is inherently batch-oriented, and wire tail latency, since wire is inherently a real-time, per-transaction path.

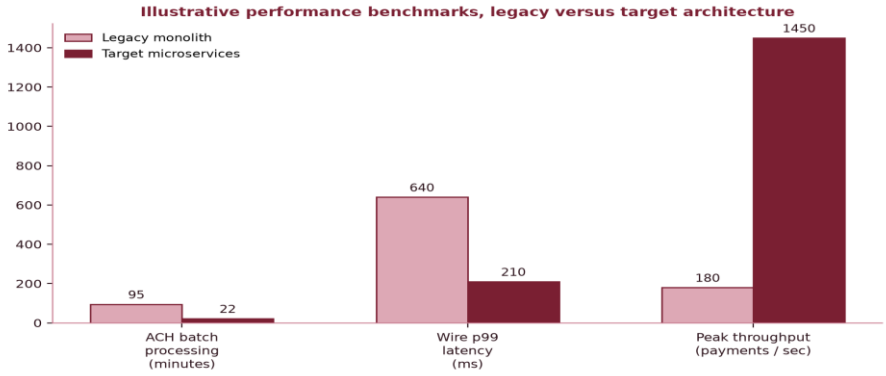


Figure.16. Illustrative performance benchmarks comparing the legacy and target architectures

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

Table.34. Benchmark and drill types used to validate the target architecture.

Test type	What it validated
ACH batch load test	Full-file processing time at peak same-day ACH volume
Wire tail latency test	p99 submission latency under concurrent load
Failover drill	Time to detect and fail over from primary to backup wire connection
Reconciliation load test	Break detection latency at full production transaction volume

XIX. CASE STUDY TIMELINE AND MILESTONES

The program was sequenced so that the lower-risk, batch-oriented ACH extraction preceded the higher-risk, real-time wire extraction, allowing the team to validate the dual-run and reconciliation tooling on ACH before applying it to the less forgiving wire rail.

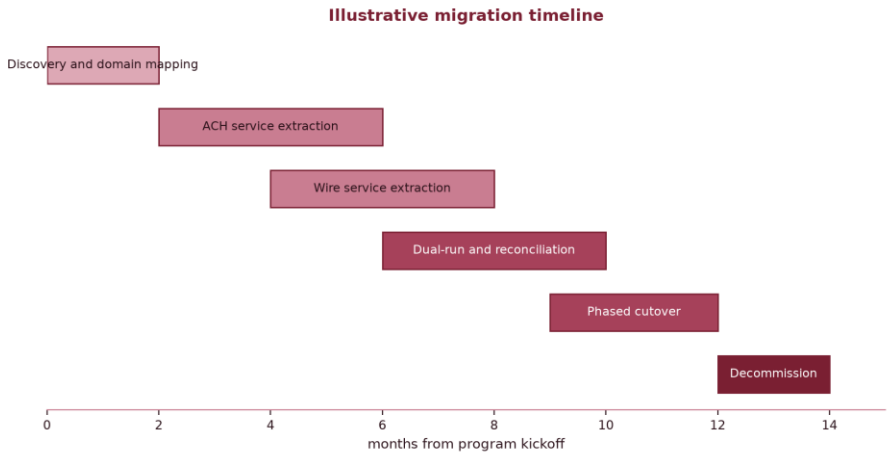


Figure.17. Illustrative migration timeline from program kickoff through legacy decommission.

Table.35.Illustrative program milestones

Milestone	Approximate timing
Domain mapping and context boundaries agreed	Month 2
ACH service reaches feature parity in dual-run	Month 6
Wire service reaches feature parity in dual-run	Month 10
Majority of ACH volume cut over	Month 12
Full cutover, both rails	Month 14
Legacy payment modules decommissioned	Month 16

XX. LESSONS LEARNED AND ANTI-PATTERNS

Table 36. Recurring anti-patterns observed during the migration program.

Anti-pattern	Symptom	Remedy
Screening bypass path	A code path posts a payment without calling the screening service	Enforce screening at a single, unavoidable choke point
Reconciliation as an afterthought	Break detection built after cutover began, not before	Build and validate reconciliation before the first traffic shift
Routing by request time instead of account	One customer's traffic bounces between old and new systems	Route deterministically by account identifier
Treating the outbox relay as fire-and-forget	Relay failures go unnoticed, events silently stop flowing	Monitor relay lag as a first-class operational metric

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

The most expensive lesson from comparable programs is almost always the same: underestimating the engineering effort required for reconciliation relative to the effort spent on the new services themselves. Teams that treat reconciliation as a lightweight validation script rather than a production-grade service tend to discover its limitations precisely when transaction volume is highest, during the majority cutover stage described in Section 17.

Table 37. Key lessons and how they were applied in this program.

Lesson	Applied to this program
Sequence by risk, not by convenience	ACH extracted before wire, despite wire being architecturally simpler
Make rollback cheap at every stage	Facade-level routing reversion kept early-stage rollback to minutes
Fund reconciliation like a core service	Reconciliation was staffed and tested to the same standard as ACH and wire services

XXI. COST AND OPERATIONAL IMPACT

Decomposition changed the shape of operating cost as much as its total level: infrastructure spend shifted from a single, coarsely scaled unit to per-service scaling matched to each rail's actual load profile, and incident volume fell steadily as the dual-run period matured.

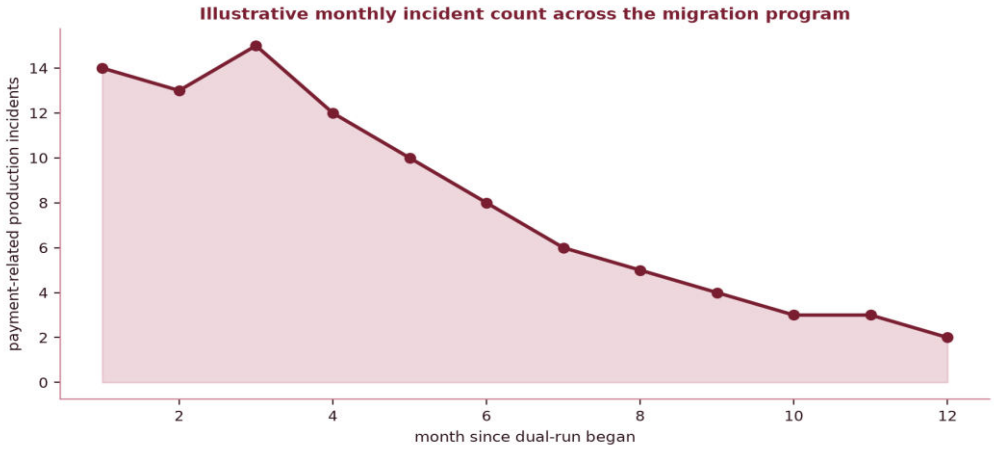


Figure.18. Illustrative monthly payment-related production incidents across the migration program

Table.38. Illustrative cost and operational comparison, legacy versus target architecture

Cost category	Legacy monolith	Target architecture
Compute scaling granularity	Single unit, sized for peak of either rail	Per-service, sized to each rail's own peak
Release cost per change	Full regression suite, every change	Scoped regression per affected service
Incident investigation effort	Manual log correlation across one large codebase	Trace-assisted correlation across services

Table.39. Illustrative operational outcomes attributed to the migration

Impact area	Illustrative outcome
Payment-related incidents per month	Declined from an initial baseline of roughly 14 to 2 by month 12 of dual-run
Time to ship a wire-only change	Reduced from a full release cycle to an independent service release
Reconciliation break rate	Declined steadily as event schema and capture lag issues were resolved

| e-ISSN: 2278 – 8875, p-ISSN: 2320 – 3765| www.ijareeie.com | Impact Factor: 8.317|| A Monthly Peer Reviewed & Referred Journal |

||Volume 12, Issue 3, March 2023||

|DOI:10.15662/IJAREEIE.2023.1203018 |

Table.40. Summary comparison of the legacy and target architectures across the dimensions covered in this case study.

Dimension	Legacy monolith	Target microservices architecture
Deployment unit	One application, both rails together	One deployable service per bounded context
Data ownership	One shared schema	One schema per service, integrated through events
Change isolation	Any change risks both rails	A rail-specific change is isolated to its own service
Regulatory audit surface	Single large codebase	Smaller, individually auditable service boundaries
Scaling model	Scaled as one unit for combined peak load	Scaled per service to its own peak load

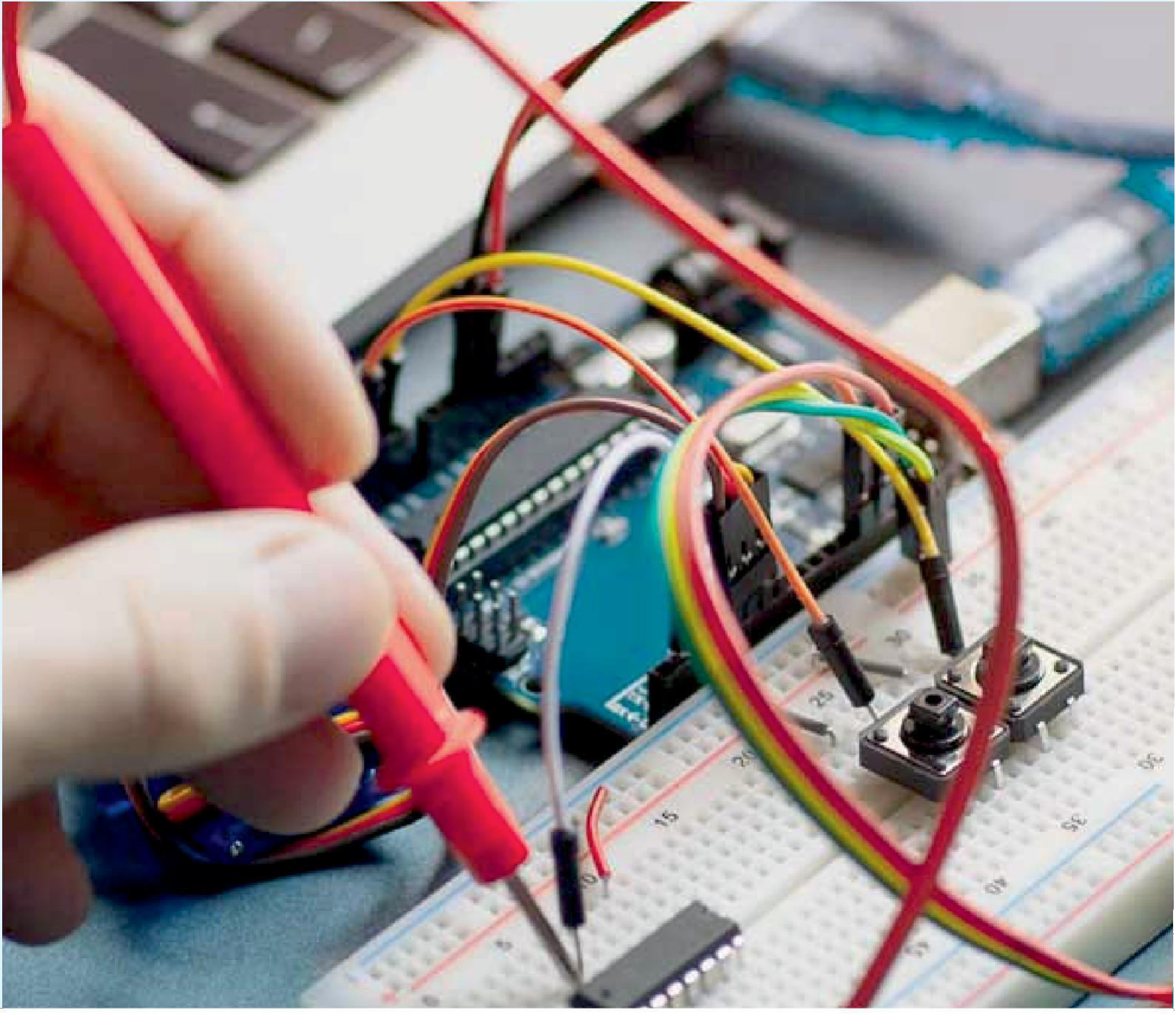
XXII. CONCLUSION

Migrating ACH and wire processing out of a shared monolith is, in the end, a story about making two different operating models, one batch-oriented and reversible, one real-time and irrevocable, architecturally legible again after years of being flattened into a single codebase. The strangler fig strategy, disciplined reconciliation, and payment-specific resiliency patterns covered in this case study are what allowed that separation to happen incrementally, under regulatory scrutiny, without a high-risk single cutover event. None of the individual techniques are unique to banking, but their sequencing and their emphasis, reconciliation funded as heavily as the services it protects, rollback made cheap at every stage, and sanctions screening enforced at a single unavoidable choke point, reflect the specific consequences of getting payment processing wrong. Teams undertaking a similar migration are well served by treating those emphases as requirements rather than as optional hardening to be added later.

A final observation concerns pace. Every stage described in Sections 4, 17, and 19 was gated on reconciliation health rather than on a calendar date, and the program's leadership treated a delayed cutover stage as a normal, expected outcome rather than as a program failure. That willingness to let evidence, not schedule pressure, decide when to advance is difficult to legislate after the fact; it is far easier to establish as a stated principle before the first migration wave begins, when the cost of slowing down is still hypothetical rather than felt.

REFERENCES

1. Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.
2. Federal Financial Institutions Examination Council. (2019). IT Examination Handbook: Business Continuity Management.
3. Federal Reserve Bank Services. (2020). Same Day ACH: Moving Payments Faster.
4. Federal Reserve Financial Services. (2021). Fedwire Funds Service: Assessment of Payment System Risk.
5. Financial Crimes Enforcement Network. (2020). Bank Secrecy Act Requirements for Financial Institutions.
6. Fowler, M. (2004). StranglerFigApplication. martinfowler.com bliki.
7. Fowler, M. (2014). CircuitBreaker. martinfowler.com bliki.
8. Fowler, M., and Lewis, J. (2014). Microservices: a definition of this new architectural term. martinfowler.com.
9. Hohpe, G., and Woolf, B. (2003). Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley.
10. Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.
11. Nacha. (2021). Nacha Operating Rules and Guidelines.
12. Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.
13. Newman, S. (2019). Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media.
14. Nygard, M. (2018). Release It! Design and Deploy Production-Ready Software, second edition. Pragmatic Bookshelf.
15. Office of Foreign Assets Control. (2019). Sanctions Compliance Guidance for the Financial Services Sector.
16. PCI Security Standards Council. (2018). Payment Card Industry Data Security Standard, version 3.2.1.
17. Richardson, C. (2018). Microservices Patterns: With Examples in Java. Manning Publications.
18. Society for Worldwide Interbank Financial Telecommunication. (2018). ISO 20022 Migration Guide.



INNO SPACE
SJIF Scientific Journal Impact Factor

Impact Factor: 8.317

International Journal of Advanced Research

in Electrical, Electronics and Instrumentation Engineering

9940 572 462 6381 907 438 ijareeie@gmail.com

Scan to save the contact details