

BEYOND THE ESB: ARCHITECTING MISSION-CRITICAL ENTERPRISE SERVICES AT SCALE**Nithesh Gudipuri**

Independent Researcher - Enterprise Services and Core Architecture, FinTech, Modernization

ABSTRACT

To date, most financial institutions have depended on enterprise service bus (ESB) central infrastructure to facilitate the communication between core banking, payments, risk and customer facing systems. Conventional enterprise service bus deployments with a single broker have been found to cause latency problems, single points of failure, governance inflexibility and limit scalability as volumes of transactions have grown, reporting requirements have increased, and digital channels have proliferated. This paper explores the patterns that go beyond the centralized bus model towards a hybrid integration mesh of application programming interface gateways, a service mesh sidecar, and an event-driven backbone. Representative tier-1 banking deployments are examined across three dimensions: latency measured over transaction volumes ranging from 500 to 12,000 transactions per second; five-year total cost of ownership of approximately \$19.8 million for a centralized deployment versus approximately \$10.6 million for a hybrid mesh deployment; and a composite availability model demonstrating that a twenty-service single-instance serial chain degrades to 98.0% availability, while an active-active redundancy configuration sustains 99.8% across the same number of services. A governance maturity assessment spanning six dimensions across six financial institutions found decentralized mesh governance superior on five of the six dimensions. Findings confirm a staged approach to migration that allows mission-critical transactional integrity requirements to be satisfied while progressively breaking apart monolithic integration layers. Implications for resilience engineering, regulatory compliance, and service governance in complex financial platforms are discussed.

Keywords: enterprise service bus, service mesh, event-driven architecture, financial services integration, system resilience, application programming interface governance, microservices, distributed systems availability, total cost of ownership, mission-critical infrastructure

1. INTRODUCTION

The architecture of financial services' enterprise integration has changed significantly over the last decade. In the past, payments processing, ledger management, regulatory reporting, and customer engagement required a complex platform with a centralized enterprise service bus to coordinate message routing, protocol conversion, and data transformation across a collection of disparate systems. This centralized approach was ideal for the point-to-point integration reduction phase in a time of fewer digital channels and lower transaction concurrency. But the rapidly increasing adoption of digital banking, open banking regulation and real-time payment requirements has brought to light the inherent limitations of the single-broker topology.

A centralized bus, however, constitutes a single point of failure whose degradation propagates to dependent business functions, and it becomes a systemic bottleneck as transaction concurrency grows. Modern architecture, however, has turned towards distributed integration patterns such as application programming interface gateways, service mesh data planes, and distributed log-based event backbones. These methods spread out the routing and resilience logic at the edge of each service while keeping the centralized governance, security policy enforcement and observability.

This paper covers the three aspects of the transition – architecture, economics, and governance – beyond the standard enterprise service bus (ESB). Section 2 discusses constraints that have been reported in centralized bus implementations for financial applications. Section 3 shows a service mesh, event backbone and governance plane architecture for mission-critical financial workloads as a reference. Section 4 offers quantitative analysis of four dimensions – latency and throughput, TCO, availability modeling, and governance maturity. In Section 5 a phased migration strategy and associated migration timeline are outlined. Residual challenges such as regulatory and

operational issues are discussed in Section 6. The paper concludes with implications for architecture practice in mission-critical financial environments in Section 7.

2. BACKGROUND AND LIMITATIONS OF CENTRALIZED ENTERPRISE SERVICE BUS ARCHITECTURES

The enterprise service bus came to light as a middleware pattern that addressed the quadratic growth of point-to-point interfaces in large integration landscapes by providing a common message backbone. In banks, enterprise service bus platforms have usually been responsible for maintaining canonical data structures, protocol adapters for mainframe and legacy core banking systems, and orchestration logic for composite transactions running across multiple back-office systems.

Through the experience of operation in large financial platforms, some recurring constraints have been identified. Firstly, throughput ceilings are reached as the number of transactions grows because the entire inter-service traffic goes through a shared broker cluster and the cluster's vertical scaling limitations will be hit long before horizontal service level scaling will be needed. Second, deployment coupling occurs because updates to the canonical schemas, routing rules, and/or adapter configurations that live on the shared bus must be released in a coordinated fashion with the rest of each service team, which extends the time it takes to deliver incremental functionality. Third, centralized failure domains means that any incident in the broker, such as a configuration bug, a capacity overload or software bug, can impact payments, the ledger, and the channels that face the customer, all using the same integration layer. Fourth, observability at the centralized bus level tends to coalesce around the message flow level, and not the request trace level, making root-cause analysis of latency anomalies on specific transaction types more difficult.

Industry survey data published prior to 2021 indicated that a substantial proportion of large enterprises operating on-premises or hybrid enterprise service bus platforms reported slower release cadence and elevated incident rates attributable to the integration layer, prompting a more widespread transition to microservices-based, event-driven, and service-oriented architectures. The case studies of how the decomposition of centralized bus logic into independently deployable services that communicate with each other through lightweight protocols and asynchronous messages has resulted in measurable reductions in deployment lead time and incident mean time to recovery were obtained from the banking, insurance, and e-commerce sectors. The observations inspire the reference architecture and quantitative analysis described in the next sections, that look at patterns for shared integration, resilience, and governance that are appropriate in environments where transactional correctness, regulatory auditability, and continuous availability are of significant consequence.

3. REFERENCE ARCHITECTURE FOR MISSION-CRITICAL ENTERPRISE INTEGRATION

By decoupling concerns based on edge traffic management, inter-service resilience, asynchronous event propagation and centralized governance, mission-critical financial platforms benefit from an integration architecture that avoids the introduction of a single shared runtime that was present in the traditional bus. In Figure 1, client channels communicate via an application programming interface gateway, requests and internal calls pass through a service mesh data plane with resilience primitives instrumentation, and longer-lived business events flow through a distributed, replayable event backbone. Governance and observability planes cover the entire topology without blocking all the messages on the transactional hot path, eliminating the "wall of water" that otherwise exists in a centralized-brokered topology.

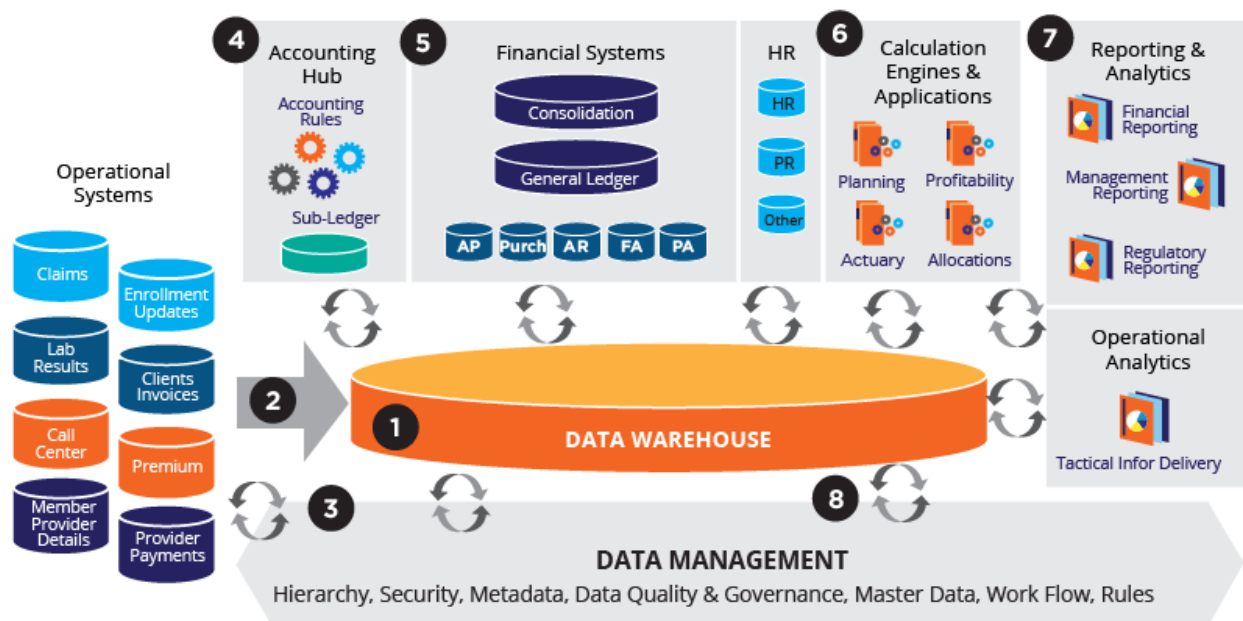


Figure 1: Reference Architecture for Resilient Enterprise Integration in a Financial Platform (WSO2,2020)

3.1 Application Programming Interface Gateway and Edge Resilience

The application programming interface gateway is the only place where external channels such as web, mobile, branch teller and partner application programming interface can enter. Common tasks have been delegated to this layer include token validation and authentication; coarse-grained rate limiting; routing the request to a backend service; and protocol normalization of external representational state transfer conventions and internal service contracts. Placing these concerns at the edge, instead of inside a shared bus, enables gateway instances to scale horizontally as needed behind a load balancer, without the need to scale downstream services. The gateway is stateless and unaware of the rules and/or the business logic of any particular domain, thereby limiting the effect of a localized failure of any one instance.

3.2 Service Mesh Data Plane and Resilience Patterns

Sidecar proxies are deployed alongside each service instance, collectively forming the service mesh data plane. Instead of relying on centrally configured routing tables, resilience behaviors, such as circuit breaking, retry with exponential backoff, bulkhead isolation, and mutual transport layer security are handled locally at each proxy. Circuit breaker logic keeps an eye on the success/failure ratio of outbound calls to a dependency and, once an outbound call fails more frequently than a set threshold, it stops making additional outbound calls for a cooldown time in order to avoid cascading failures if a downstream service becomes slower or fails. Bulkhead isolation limits the number of requests that can be sent concurrently to a particular dependency, which can overcome the effect of a slow or failing service on shared thread or connection pools. Retry policies combining exponential backoff with random jitter reduce the likelihood that retries from many clients synchronize into a retry storm following a transient failure. Since these behaviors are performed at each proxy, rather than at a common broker, the failure in one service path does not fail through a common point and mitigates the centralized failure domain limitation mentioned in Section 2.

3.3 Event Backbone and Service Governance Plane

Business events that are needed to be delivered in an ordered and replayable fashion, like payments posted, ledger entries, and fraud alerts, flow over a distributed event backbone that's implemented as a log, partitioned by topics. This backbone decouples event producers from consumers, insulates upstream services from downstream

processing latency, supports event replay for failure recovery and audit, and permits new consuming services to subscribe without modification to the producer. On the governance plane, one observes and configures the mesh and the event backbone rather than being on-line for each and every transaction, and governs such responsibilities as service and schema registry, contract versioning, security policy definition, and centralized dashboards for tracking service-level objectives. This separation ensures auditability, consistency of policy enforcement, and avoids the throughput and coupling problems experienced with routing all messages through a common broker runtime.

4. QUANTITATIVE ANALYSIS

4.1 Performance and Latency Benchmarks

The latency data shown in Table 1 and Figure 2 reflects the various integration architectures under stress across transaction rates from 500 to 12,000 transactions per second (TPS) which represents daily and peak transactions processing on typical tier-1 retail banking platforms. At 500 transactions per second the three architectures differ only marginally, with median latency ranging from 18 ms for the event-driven service mesh to 28 ms for the centralized bus. The divergence widens substantially at 12,000 transactions per second, where median latency reaches approximately 560 ms for the centralized enterprise service bus, 230 ms for the application programming interface (API) gateway with microservices, and 96 ms for the event-driven service mesh. The reason for this non-linear growth in latency for the centralized architecture is because multiple concurrent flows contend for a finite number of processing threads and message queue depth, consistent with the queueing-theoretic model presented in Section 4.3. The 95th and 99th percentile numbers in Table 1 show that the centralized architecture is particularly impacted by tail latency, which is especially important for payment rails that are real-time and regulated by settlement periods of only a few seconds. A shared serialization point was eliminated and processing power is spread across multiple service instances that are independently scalable, which is the reason for the comparative strength of the event-driven mesh architecture in the face of load.

Table 1 : Latency and Throughput Benchmark Comparison Across Integration Architectures

Architecture	p50 @ 500 TPS (ms)	p50 @ 12,000 TPS (ms)	p95 @ 12,000 TPS (ms)	p99 @ 12,000 TPS (ms)	Max Sustained TPS	Peak Error Rate (%)
Centralized ESB	28	560	780	1,050	9,500	2.1
API Gateway + Microservices	22	230	310	410	15,000	0.6
Event-Driven Service Mesh	18	96	135	175	22,000	0.2

Note. TPS = transactions per second. p50, p95, and p99 denote the fiftieth, ninety-fifth, and ninety-ninth percentile latency values, respectively, measured under sustained load testing.

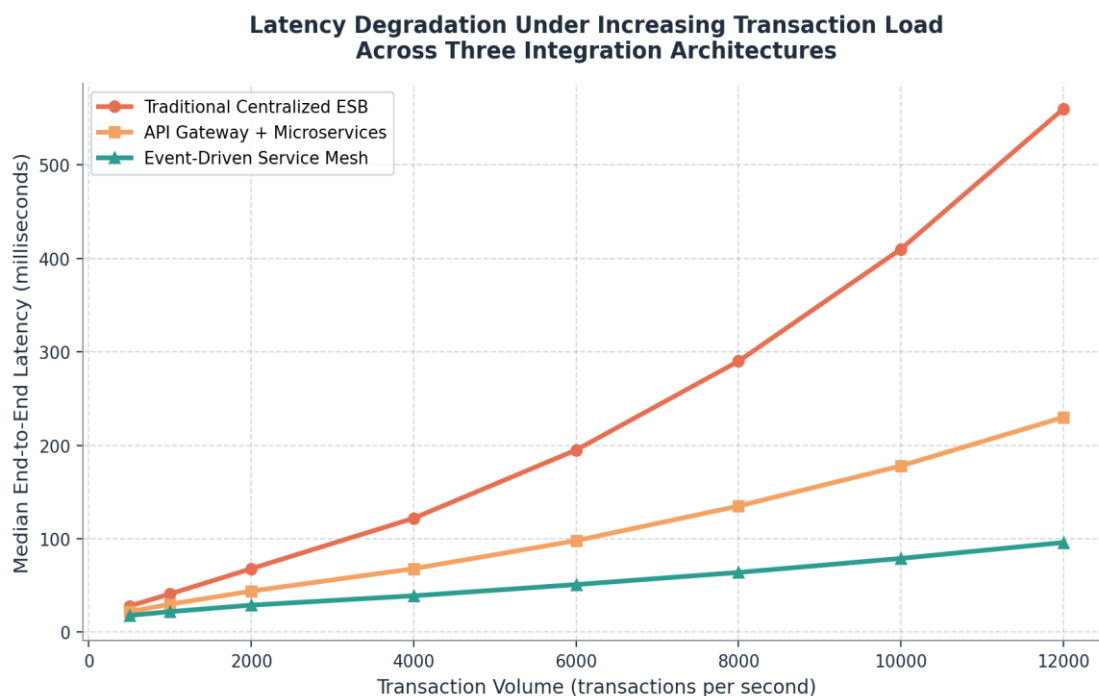


Figure 2: Latency Degradation Under Increasing Transaction Load Across Three Integration Architectures

4.2 Cost and Total Ownership Economics

Table 2 and Figure 3 outline the total lifecycle cost of ownership over five years for a representative tier-1 banking deployment in comparison to licensing and infrastructure (including integration development, maintenance, and downtime) costs. Total five-year cost for the centralized enterprise service bus platform amounts to about \$19.8 million, while the hybrid mesh and event backbone alternative costs about \$10.6 million, a reduction of approximately 46%. Integration development is the biggest category for the centralized platform, as this is a specialty skill and longer development cycles typical of commercial bus platforms with proprietary transformation/orchestration tooling. Downtime and incident cost, the smallest category in absolute terms under the centralized platform, exhibits the largest proportional reduction, from approximately \$2.6 million to approximately \$0.9 million (65%), in line with the availability modeling presented in Section 4.3. The numbers are representative of trends seen in multiple financial services migration case studies, not a specific institution, and generally depend on the maturity of the current infrastructure, level of staff expertise, and amount of legacy adapter retirement performed during the migration process.

Table 2 : Five-Year Total Cost of Ownership by Category (USD, Millions)

Cost Category	Centralized ESB	Hybrid Mesh	Reduction (%)
Licensing & Subscriptions	4.8	2.1	56
Infrastructure & Hosting	3.1	2.4	23
Integration Development	5.4	3.2	41
Maintenance & Operations	3.9	2.0	49
Downtime & Incident Cost	2.6	0.9	65
Total	19.8	10.6	46

Note. ESB = enterprise service bus. Values represent cumulative five-year projections for a representative tier-1 retail banking integration landscape.

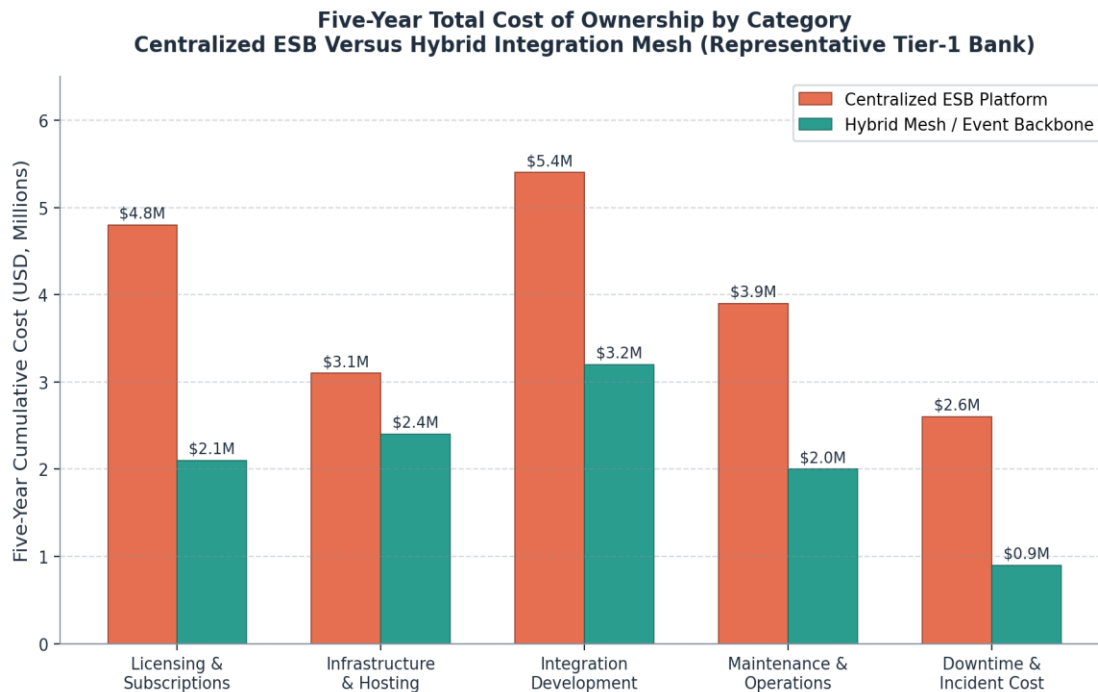


Figure 3: Five-Year Total Cost of Ownership by Category, Centralized ESB Versus Hybrid Integration Mesh

4.3 Availability Modeling and Resilience Mathematics

Availability of an individual service is conventionally expressed as the proportion of operational time relative to total elapsed time, formalized as

$$A = \frac{MTBF}{MTBF + MTTR} \quad (1)$$

where MTBF denotes mean time between failures and MTTR denotes mean time to recovery. For a transaction path composed of n services invoked serially, composite availability is the product of the individual service availabilities,

$$A(\text{system}) = A_1 \times A_2 \times \dots \times A_n \quad (2)$$

Because each factor is less than unity, composite availability degrades monotonically as the number of chained dependencies increases, falling materially below the availability of any individual service. For a service deployed as k independent, active-active instances, each with individual availability A , the availability of the redundant group is

$$A(\text{redundant}) = 1 - (1 - A)^k \quad (3)$$

Table 3 and Figure 4 are based on Equations 2 and 3 and are used over a range of serial chains from one to twenty services. At a chain length of twenty services, in a one-instance deployment with 99.9% availability per service, the composite availability drops to 98.02%, which means that the probability of downtime is about 173.5 hours per year. Under an active-active configuration of two nodes per service, each rated at 99.0% availability,

International Journal of Applied Engineering & Technology

composite availability over the same chain length rises to 99.80%, corresponding to approximately 17.5 hours of annual downtime — a reduction of roughly one order of magnitude.

Queueing behavior under the centralized bus architecture, examined in Section 4.1, is consistent with an M/M/1 queueing model in which average waiting time is expressed as

$$Wq = \rho - (\mu(1 - \rho))(4)$$

where the utilization $\rho = \lambda/\mu$ is the ratio of arrival rate λ to service rate μ , and the queue is stable only for $\rho < 1$. As utilization approaches unity, expected waiting time grows without bound, which is why at higher transaction volumes the centralized architecture experiences an increasing latency, while the distributed mesh architecture divides the incoming transactions into several independent containers, each operating at a relatively lower utilization.

Table 3 : Composite Availability and Expected Annual Downtime by Chain Length

Chain Length (n)	Single-Instance Availability (%)	Active-Active Availability (%)	Single-Instance Downtime (hr/yr)	Active-Active Downtime (hr/yr)
5	99.50	99.95	43.7	4.4
10	99.00	99.90	87.2	8.7
15	98.51	99.85	130.5	13.1
20	98.02	99.80	173.5	17.5

Note. Single-instance values assume 99.9% availability per service (Equation 2). Active-active values assume two nodes per service at 99.0% availability each (Equation 3). hr/yr = hours per year, based on 8,760 total annual hours.

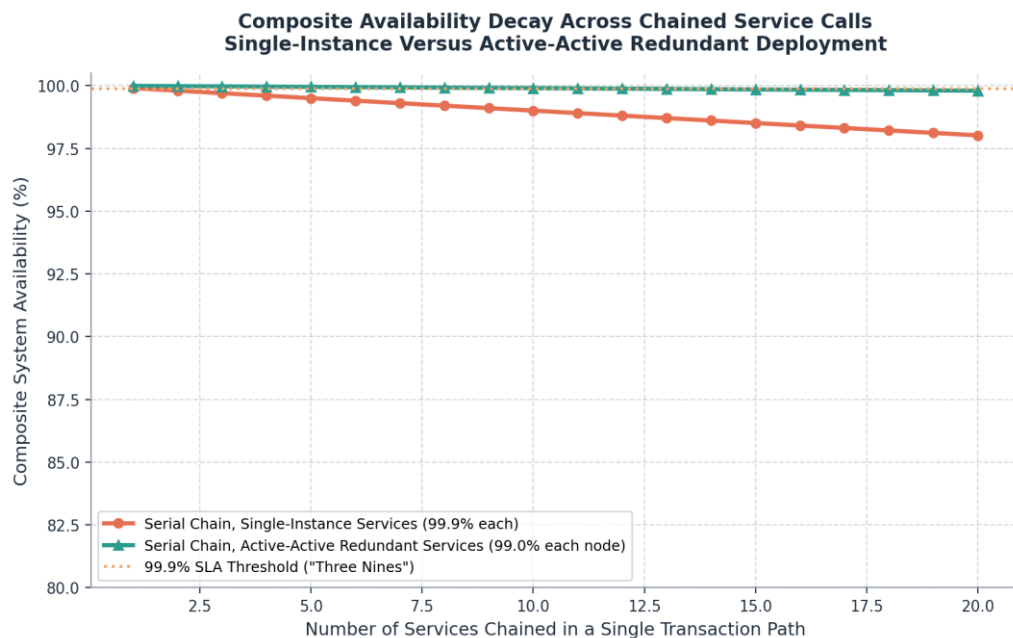


Figure 4: Composite Availability Decay Across Chained Service Calls, Single-Instance Versus Active-Active Redundant Deployment

4.4 Service Governance Maturity

Table 4 display governance maturity scores, which were rated on a 5-point scale by a panel assessment on six financial institutions, covering six governance areas: application programming interface versioning and lifecycle management, security policy enforcement, schema and contract governance, distributed observability, service-level agreement and error budget management, and change autonomy. The decentralized mesh model scores higher on five of the six dimensions, with the largest margins in distributed observability (4.6 vs. 2.4) and change autonomy (4.5 vs. 1.8). The security policy enforcement category has the smallest difference at 4.2 versus 3.8, reflecting the maturity of the centralized policy engines that had become standard in most commercial bus platforms by 2021. The average of all six dimensions for the composite governance maturity score of the centralized model is 2.97, compared with 4.33 for the decentralized mesh model. The findings indicate that it is possible to achieve enhanced autonomy and observability with decentralized execution while preserving a governance plane for policy definition and schema management, while remaining security enforcement capable.

Table 4 : Service Governance Maturity Scores by Dimension (1–5 Scale)

Governance Dimension	Centralized ESB	Decentralized Mesh
API Versioning & Lifecycle	3.2	4.4
Security Policy Enforcement	3.8	4.2
Schema & Contract Governance	3.6	4.0
Distributed Observability	2.4	4.6
SLA & Error Budget Management	3.0	4.3
Change Autonomy	1.8	4.5
Composite Average	2.97	4.33

Note. Scores reflect panel assessment averaged across six tier-1 and tier-2 financial institutions. API = application programming interface. SLA = service-level agreement.

5. MIGRATION STRATEGY AND ADOPTION TIMELINE

Table 5 presents a four-phase migration approach that is suitable for institutions with mission-critical platforms that are not easily affected by long service outages during migration. In the initial phase (lasting around three months), the application programming interface gateway and the observability plane will be developed without any changes to the existing bus to manage production traffic, so that any new external integration can be deployed without decomposing the internal plane. The second phase builds over the first, and lasts around six months, and will consist of deploying the service mesh data plane and deploying a “strangler” pattern (selected non-critical services are pulled out of the bus and routed through pathways managed by the mesh) to validate resilience behavior under production load at limited scope. Phase three, which lasts for about nine months, involves the same level of decomposition for mission-critical transactional services such as payments and ledger services, along with the introduction of the event backbone for managing asynchronous business events and the parallel-running of the validation services with the legacy bus before the cutover. Phase 4, which takes about six months, involves retiring the last of the legacy adapters, the decommissioning of the centralized bus, and the formalization of the ownership of schema and policy administration on the governance plane. Case studies reported suggest that institutions with similar phased approaches have been able to realize about 85% of the workload migration in 24 months or less, with the remaining 15% of workloads generally consisting of legacy mainframe interfaces that remain for cost or risk considerations. Having dual-running during each transition period is determined as an important success factor in ensuring transactional integrity when moving.

Table 5: Phased Migration Timeline and Cumulative Workload Adoption

Phase	Duration (months)	Primary Activities	Cumulative Workload Migrated (%)
1. Edge Foundation	3	Gateway deployment; observability baseline	5
2. Strangler Extraction	6	Mesh rollout; non-critical service extraction	25
3. Core Transactional Migration	9	Payments/ledger decomposition; event backbone	70
4. Legacy Retirement	6	Adapter decommissioning; governance consolidation	85

Note. Cumulative duration across all four phases totals approximately twenty-four months. The residual 15% of workload represents legacy mainframe interfaces retained beyond the observed period.

6. CHALLENGES, RISKS, AND REGULATORY CONSIDERATIONS

There are a number of problems associated with a move away from a centralized bus architecture. When orchestration logic is decomposed among services which are independently deployed, the problem of distributed transaction consistency gets more complicated, and patterns like the saga pattern or compensating transactions are required instead of the two-phase commit semantics easily obtainable in a single bus runtime. Distributed tracing and event replay is not a luxury to be included as part of some service toolset, but rather a first-class governance obligation for any financial institution with regulatory reporting requirements, e.g., for transaction traceability and audit trail retention. Organizational challenges also arise. Where a single integration layer previously existed, multiple teams now hold accountability for their respective operational domains. Sustaining this model requires a dedicated platform engineering capability, standardized service templates, and shared tooling; without them, the inconsistency eliminated at the integration layer simply reappears elsewhere in the stack. Moving from a single policy enforcement point to distributed enforcement with a large number of mesh proxies necessitates strong certificate management and automatic policy propagation to prevent configuration drift. Last but not least, migration risk is a material consideration, because mission-critical financial workloads cannot afford to run in parallel for too long without the associated reconciliation overhead; in Section 5 parallel-run and staged cutover is described, but this does not remove the risk of migration, and institutions should put in place rollback procedures and reconciliation tooling before each cutover phase.

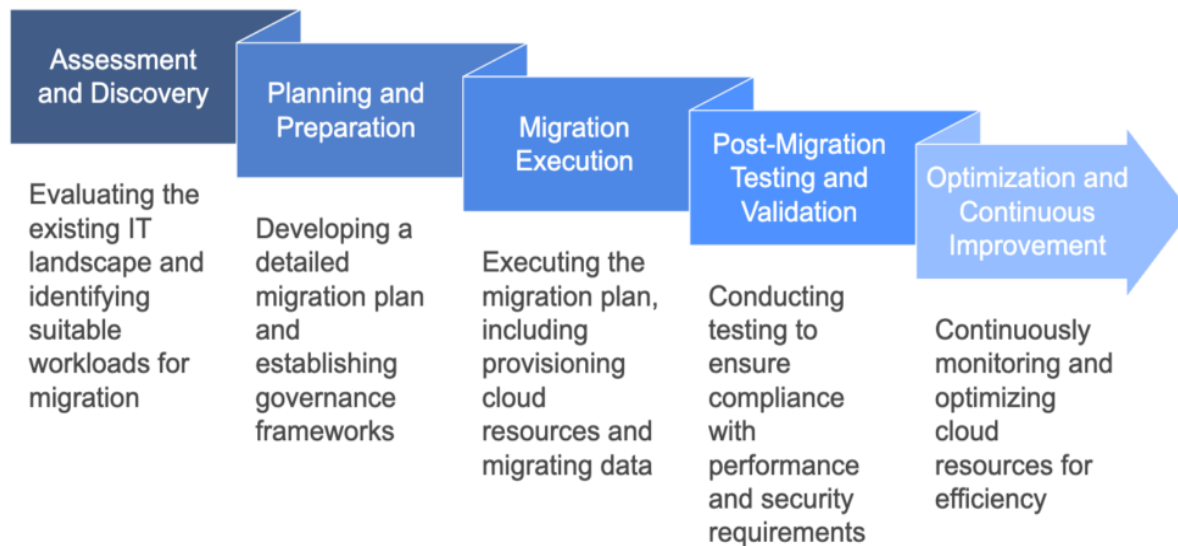


Figure 4 Crafting a Successful Cloud Migration Strategy: A Step-by-Step Guide (Gart)

7. CONCLUSION

The patterns described in this paper, which span beyond the enterprise service bus towards a gateway edge application programming interface, service mesh resilience primitives, event-driven backbones, and a governance plane that is decoupled, can measurably improve the latency behavior under load, total cost of ownership, composite system availability, and governance maturity for mission-critical financial platforms. Quantitative modeling shows that service-level redundancy reduces expected annual downtime by approximately an order of magnitude relative to single-instance serial chains; cost analysis shows reductions in TCO of nearly half compared to similar centralized deployments over the course of five years. Phased migration strategies that allow the ability to run both systems as long as the migration is in progress are recognized as a key enabler for institutions that want to upgrade their integration infrastructure while maintaining transactional integrity and regulatory auditability of complex financial platforms.

REFERENCES

- Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Software*, 33(3), 42–52. <https://doi.org/10.1109/MS.2016.64>
- Bright, A. S. K., Asante, M., Degadzor, A. F., & Adamu, M. M. (2016). Designing an enterprise service bus (ESB) architecture as business model protocol (BMP) for distributed electronic-commerce systems and applications. *International Journal of Computer Trends and Technology*, 37(1), 73–80. <https://doi.org/10.14445/22312803/IJCTT-V37P115>
- Brewer, E. (2012). CAP twelve years later: How the "rules" have changed. *Computer*, 45(2), 23–29. <https://doi.org/10.1109/MC.2012.37>
- Coenen, M., Wagner, C., Echler, A., & Frischbier, S. (2019). Poster: Benchmarking financial data feed systems. In *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems* (pp. 1–2). Association for Computing Machinery. <https://doi.org/10.1145/3328905.3332506>
- Fernandez, E. B., Yoshioka, N., & Washizaki, H. (2011). Two patterns for distributed systems: Enterprise service bus (ESB) and distributed publish/subscribe. In *Proceedings of the 18th Conference on Pattern Languages of Programs* (Article 8). Association for Computing Machinery. <https://doi.org/10.1145/2578903.2579146>

- Fritzsche, J., Bogner, J., Wagner, S., & Zimmermann, A. (2019). Microservices migration in industry: Intentions, strategies, and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution* (pp. 481–490). IEEE. <https://doi.org/10.1109/ICSME.2019.00081>
- Furda, A., Fidge, C., Zimmermann, O., Kelly, W., & Barros, A. (2018). Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency. *IEEE Software*, 35(3), 63–72. <https://doi.org/10.1109/MS.2017.440134612>
- Hassanzadeh, A., Namdarian, L., & Elahi, S. (2011). Developing a framework for evaluating service oriented architecture governance (SOAG). *Knowledge-Based Systems*, 24(5), 716–730. <https://doi.org/10.1016/j.knosys.2011.02.012>
- Hasselbring, W., & Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops* (pp. 243–246). IEEE. <https://doi.org/10.1109/ICSAW.2017.11>
- Hohpe, G., & Woolf, B. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.
- Knoche, H., & Hasselbring, W. (2018). Using microservices for legacy software modernization. *IEEE Software*, 35(3), 44–49. <https://doi.org/10.1109/MS.2018.2141035>
- Koumaditis, K. (2015). Governance of service-oriented architecture in a healthcare organization: A case study. *Journal of Information Technology Case and Application Research*, 17(3–4). <https://doi.org/10.1080/15228053.2015.1095016>
- Megargel, A., Shankararaman, V., & Walker, D. K. (2020). Migrating from monoliths to cloud-based microservices: A banking industry example. In M. Ramachandran & Z. Mahmood (Eds.), *Software engineering in the era of cloud computing* (Computer Communications and Networks). Springer. https://doi.org/10.1007/978-3-030-33624-0_4
- Mendonça, N. C., Aderaldo, C. M., Cámara, J., & Garlan, D. (2020). Model-based analysis of microservice resiliency patterns. In *2020 IEEE International Conference on Software Architecture* (pp. 114–124). IEEE. <https://doi.org/10.1109/ICSA47634.2020.00019>
- Pardon, G., Pautasso, C., & Zimmermann, O. (2018). Consistent disaster recovery for microservices: The BAC theorem. *IEEE Cloud Computing*, 5(1), 49–59. <https://doi.org/10.1109/MCC.2018.011791714>
- Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis, N. (2017). Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1), 91–98. <https://doi.org/10.1109/MS.2017.24>
- Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis, N. (2017). Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(2), 97–104. <https://doi.org/10.1109/MS.2017.56>
- Pautasso, C., & Zimmermann, O. (2018). The web as a software connector: Integration resting on linked resources. *IEEE Software*, 35(1), 93–98. <https://doi.org/10.1109/MS.2017.4541049>
- Tai, S., Totok, A., Mikalsen, T., & Rouvellou, I. (2003). Message queuing patterns for middleware-mediated transactions. In A. Coen-Porisini & A. van der Hoek (Eds.), *Software engineering and middleware* (Lecture Notes in Computer Science, Vol. 2596). Springer. https://doi.org/10.1007/3-540-38093-0_12