2581-4575
International Journal For Recent Developments in Science & Technology
Crossref
A Peer Reviewed Research Journal



Monolithic, Modular-Monolithic and Microservices Architectures for Financial Software: A Capability-Level Decision Framework for Consistency, Resilience and Change Velocity

Dr. Satya Venkata Surya Narayana Murthy
Mphasis Corporation
41 Madison Ave 35th floor, New York, NY 10010
nimusha73@gmail.com

Abstract

Financial institutions must process high transaction volumes, protect sensitive data, satisfy regulatory obligations and change their digital services quickly. Choosing between monolithic, modular-monolithic and microservices architectures is therefore a business and governance decision as much as a technical one. This paper proposes a capability-level decision framework that applies the choice separately to each business capability rather than to an entire platform. The framework combines three gating questions (strict transactional consistency, uneven scaling or independent release needs, and organizational DevOps maturity) with a weighted multi-criteria score across nine dimensions: throughput, latency stability, scalability, resilience, data consistency, security, auditability, deployment agility and operational complexity. It examines how cross-service transactions can be handled with sagas, the transactional outbox and idempotency, and how delivery-performance measures can indicate readiness for decomposition. Applied to representative financial scenarios, the framework recommends modular-monolithic cores for strongly consistent ledger and authorization functions, microservices for elastic channels, fraud analytics, reporting and notifications, and staged hybrid modernization when platform maturity is still developing. The study is conceptual and does not report production benchmarks; it provides a transparent, repeatable decision procedure and evaluation criteria for future empirical validation.

International Journal For Recent Developments in Science & Technology
2581-4575
Crossref
A Peer Reviewed Research Journal



Keywords: microservices, modular monolith, software architecture decision-making, financial software, data consistency, sagas, DevOps, operational resilience.

1. Introduction

Banking, payment, insurance, trading and risk platforms must remain correct, secure and available while adding digital products at an increasing pace. Regulators now expect firms to withstand and recover from operational disruptions [21], and payment messaging is converging on rich data standards such as ISO 20022 [8]. Architecture choices shape all of these outcomes.

Many institutions still run monolithic systems in which presentation, business logic, data access and integration form one deployable unit. These systems are stable and easy to govern but can become hard to scale, change and recover. Microservices decompose capabilities into independently deployable services [1], [2], yet they introduce distributed-system complexity: partial failures, network latency, schema evolution and difficult cross-service consistency [7], [19]. A third option, the modular monolith, keeps a single deployment while enforcing internal module boundaries [5].

Most guidance treats the choice as platform-wide. In practice, a ledger, a fraud engine and a notification service have very different needs. This paper therefore asks: **how should a financial institution choose an architecture for each business capability, given its consistency requirements, workload pattern and organizational maturity?** Its contributions are:

1) A capability-level decision procedure with three gating questions (Fig. 1).
2) A weighted multi-criteria score across nine dimensions to rank candidate architectures transparently.
3) Guidance on cross-service consistency (sagas, outbox, idempotency) and on measuring organizational readiness.
4) A decision matrix for common financial scenarios and a staged modernization path.

International Journal For Recent Developments in Science & Technology
2581-4575
Crossref
A Peer Reviewed Research Journal


2. Background and Related Work

A. Architectural Styles

Lewis and Fowler defined microservices as independently deployable services organized around business capabilities [1]; Newman [2], Richardson [10] and Richards and Ford [5] describe patterns and trade-offs, including the modular monolith. Domain-driven design supplies the bounded contexts that define good service or module boundaries [6], and Conway's law reminds us that system structure mirrors organizational structure [14].

B. Empirical Evidence on Microservices

Surveys and reviews document both benefits and pains. Dragoni *et al.* trace the evolution and open problems of microservices [15]; Soldani *et al.* synthesize practitioner reports of gains (scalability, independent deployment) and pains (complexity, testing, data management) [19]; Di Francesco *et al.* map academic research [18]; and migration studies report that organizational readiness is a key success factor [16], [17].

C. Consistency in Distributed Data

Distributed systems trade consistency against availability under network partitions [12], [24]. Sagas replace long distributed transactions with sequences of local transactions and compensations [11], and the transactional outbox and messaging patterns keep database updates and events aligned [4], [7], [10].

D. Delivery Performance and Resilience

Forsgren *et al.* identified delivery-performance measures (deployment frequency, lead time, change-failure rate and time to restore) that correlate with organizational performance [13]. Chaos engineering tests resilience proactively [23], and tail latency becomes critical as call graphs deepen [22]. Container orchestration supports automated recovery and scaling [3].

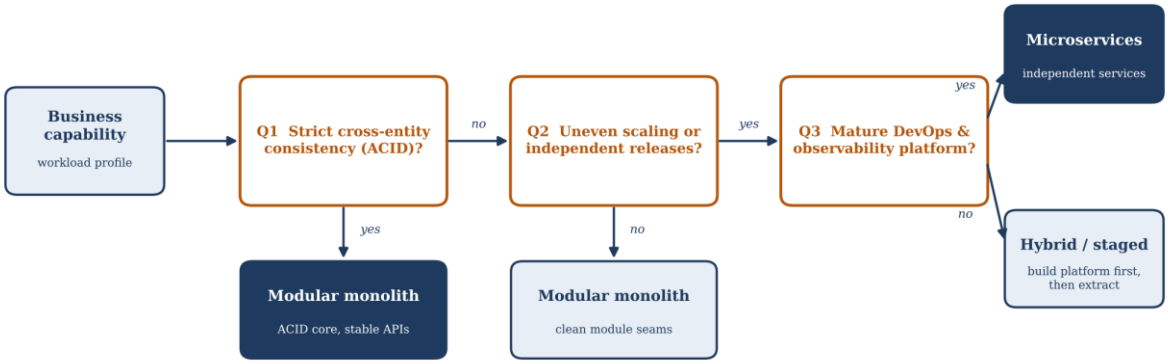
Gap. Existing guidance is largely platform-level and qualitative. A transparent, capability-

International Journal For Recent Developments in Science & Technology
2581-4575
Crossref
A Peer Reviewed Research Journal

level procedure that includes consistency and organizational maturity is missing for regulated financial systems.

3. Methodology

The study uses a design-oriented, qualitative-comparative method. The framework was derived by synthesizing the architectural literature above and was applied to representative



Decision applied per business capability, so one institution may legitimately combine several outcomes

financial workloads: account transfers, payment processing, fraud scoring, compliance reporting, transaction logging, notifications and customer service. Supporting factors such as JVM behavior, event-driven messaging, API governance, container orchestration, CI/CD and monitoring are considered because they strongly influence whether either style succeeds. Findings are decision guidance, not statistically measured superiority.

4. The Decision Framework

A. Gating Questions

Each business capability is passed through three gating questions (Fig. 1). Strict cross-entity consistency points to a modular-monolithic core; otherwise, uneven scaling or independent release needs justify decomposition, but only where the DevOps and observability platform is mature. Where it is not, a staged hybrid path builds the platform first.

International Journal For Recent Developments in Science & Technology
2581-4575
Crossref
A Peer Reviewed Research Journal



Fig. 1. Capability-level decision procedure. Each business capability passes through three gating questions; one institution may combine several outcomes.

B. Multi-Criteria Score

Where gating does not settle the choice, candidate architectures a are ranked by a weighted score across dimensions d, where s_{a,d} ∈ [1, 5] is an evidence-based rating and the weights w_d reflect the capability's priorities:

$$U(a) = \Sigma_d\, w_d \cdot s_{a,d}, \quad \Sigma_d\, w_d = 1 \qquad (1)$$

Table I gives the nine dimensions with the typical rating pattern of each style. Weights differ by capability: a ledger weights consistency and auditability highly, while a digital channel weights scalability and agility.

TABLE I. Evaluation dimensions and typical characteristics

Dimension	**Monolith / modular monolith**	**Microservices**	**Financial interpretation**
Throughput and scalability	Scaled as one unit	Selected services scale horizontally	Valuable for payment peaks and fraud workloads
Latency stability	In-process calls; predictable on short paths	Network hops add latency and tail risk [22]	Matters most for authorization paths
Resilience	A defect can affect the whole process	Isolation limits blast radius	Requires strong observability

International Journal For Recent Developments in Science & Technology
2581-4575
Crossref
A Peer Reviewed Research Journal

Data consistency	Single ACID database	Service-owned data; sagas and eventual consistency [11]	Ledgers usually stay centralized
Security and auditability	Centralized controls and audit	Granular policies; needs uniform trace IDs	Standard logging and identity propagation are essential
Deployment agility	Coordinated releases	Independent CI/CD per service	Faster change with stronger governance
Operational complexity	Lower at small scale	Higher platform, network and monitoring load	Value only with platform maturity

C. Availability of Decomposed Paths

Decomposition changes availability. For a serial path of k services with availability A_i and r independent replicas each:

$$A_{path} = \Pi_{i=1..k} [\, 1 - (1 - A_i)^r \,] \qquad (2)$$

Without replication, adding services lowers path availability; with replication and isolation of non-critical services, the critical path can become more available than a single process.

5. Cross-Service Consistency

Consistency requirements should shape boundaries. Where one business operation must update several entities atomically, such as debiting one account and crediting another, those entities belong in the same module or service. Where a workflow can tolerate brief intermediate states, three patterns make decomposition safe:

- **Sagas** split the workflow into local transactions, each with a compensating action if a later step fails [11].

International Journal For Recent Developments in Science & Technology
2581-4575
Crossref
A Peer Reviewed Research Journal

- **Transactional outbox** writes the business change and the outgoing event in one local transaction, then publishes the event reliably [10].
- **Idempotency keys** ensure that retried or duplicated messages produce exactly one financial effect.

Independent reconciliation then verifies balances end to end. These patterns add design and testing effort, which the multi-criteria score should reflect under operational complexity.

6. Organizational Readiness

Architecture cannot be separated from organizational capability [14], [16]. Microservices require automated testing, CI/CD, infrastructure as code, standard observability, clear service ownership and incident response. Delivery-performance measures [13] give an objective readiness signal. One useful indicator is change coupling, the share of changes that require coordinated deployment of more than one unit:

$$CC = |\{ \text{changes touching} > 1 \text{ deployable unit} \}| \,/\, |\text{changes}| \qquad (3)$$

High change coupling in a monolith suggests a module is a good candidate for extraction; high coupling between microservices suggests boundaries are wrong. Institutions with low deployment frequency or long restore times should first strengthen the platform, which is the hybrid path in Fig. 1.

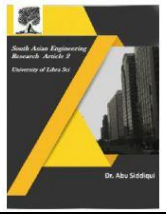
7. Decision Matrix for Financial Scenarios

Applying the framework to common scenarios gives the recommendations in Table II.

TABLE II. Recommended architecture by financial scenario

Scenario	Recommended architecture	Reason	Key controls

International Journal For Recent Developments in Science & Technology
2581-4575
Crossref
A Peer Reviewed Research Journal



Core ledger posting and deterministic authorization	Modular monolith or one well-bounded service	Strong consistency and predictable latency dominate	ACID boundaries, transaction audit, rollback testing
Digital payment channels with peak surges	Microservices	Independent horizontal scaling protects availability	API gateway, rate limits, autoscaling
Fraud detection and risk scoring	Microservices with event streaming	Compute-intensive models evolve independently	Model governance, event tracing, data lineage
Compliance reporting and monitoring	Event-driven services	Audit workloads scale separately from transactions	Immutable logs, retention policy, regulatory mapping
Institution with limited DevOps maturity	Modular monolith first, then staged extraction	Operational simplicity outweighs distribution benefits	Clean module seams, migration roadmap

8. Modernization Recommendations

- Modernize incrementally, extracting capabilities around a stable core in the manner of the strangler-fig pattern [20], and only where a measurable objective justifies it.
- Map capabilities and rank them by scaling pressure, change frequency and consistency needs before drawing boundaries [6].
- Standardize runtime baselines, container images, logging, identity, encryption, pipelines and security scanning across all services [9].
- Establish distributed tracing, service-level objectives, automated recovery and disaster-recovery and chaos testing before large-scale decomposition [21], [23].
- Track total cost of ownership: flexibility gains can be offset by platform and operational costs.



9. Limitations and Future Research

The framework is conceptual and has not been validated with production benchmarks from regulated institutions. Ratings and weights in Eq. (1) involve expert judgment and should be elicited systematically. Future work will validate the procedure through case studies, quantify cost models and test distributed-transaction strategies and resilience with controlled experiments and chaos engineering.

10. Conclusion

Monolithic, modular-monolithic and microservices architectures all remain relevant to financial software. By making the choice per business capability and gating it on consistency, workload pattern and organizational maturity, institutions can keep strongly consistent cores where determinism matters and decompose only where scalability, resilience or change velocity justify distributed-system complexity. The most effective result is often a hybrid: a governed core surrounded by independently scalable services.

Acknowledgment

Technology and product names are used for identification and historical context only; no endorsement by their respective owners is implied.

References

1. J. Lewis and M. Fowler, “Microservices: A definition of this new architectural term,” martinfowler.com, Mar. 2014.
2. S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2021.
3. B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*, 3rd ed. Sebastopol, CA, USA: O’Reilly Medias
4. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley, 2003.

5. M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, 2020.
6. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA, USA: Addison-Wesley, 2003.
7. M. Kleppmann, *Designing Data-Intensive Applications*. Sebastopol, CA, USA: O'Reilly Media, 2017.
8. *Financial Services—Universal Financial Industry Message Scheme—Part 1: Metamodel*, ISO 20022-1:2013, International Organization for Standardization, Geneva, Switzerland, 2013.
9. Joint Task Force, "Security and privacy controls for information systems and organizations," NIST Special Publication 800-53 Rev. 5, National Institute of Standards and Technology, Gaithersburg, MD, USA, Sep. 2020.
10. C. Richardson, *Microservices Patterns: With Examples in Java*. Shelter Island, NY, USA: Manning, 2018.
11. H. Garcia-Molina and K. Salem, "Sagas," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, San Francisco, CA, USA, 1987, pp. 249–259.
12. E. Brewer, "CAP twelve years later: How the 'rules' have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.
13. N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps*. Portland, OR, USA: IT Revolution Press, 2018.
14. M. E. Conway, "How do committees invent?" *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.
15. N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017, pp. 195–216.
16. A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.
17. D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.
18. P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *J. Syst. Softw.*, vol. 150, pp. 77–97, 2019.
19. J. Soldani, D. A. Tamburri, and W.-J. van den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 146, pp. 215–232, 2018.
20. M. Fowler, "StranglerFigApplication," martinfowler.com, Jun. 2004.
21. Basel Committee on Banking Supervision, "Principles for operational resilience," Bank for International Settlements, Basel, Switzerland, Mar. 2021.
22. J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.
23. A. Basiri *et al.*, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
24. W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009.