RESEARCH ARTICLE OPEN ACCESS

# Incremental Core Transformation: A Modular First Approach to Legacy Modernization in Banking

Ashish G. Vishwakarma
Senior Software Engineer, JPMorgan Chase Co.

**ABSTRACT**
Legacy banking systems built on COBOL mainframes face pressure to modernize for agility, cost-efficiency, and compliance. However, their transformation is high-risk due to scale and the need for continuous availability. This paper presents the Modular-First Legacy Modernization (MFLM) framework, integrating SEI's System Analysis and Migration (SAM), OMG's Architecture-Driven Modernization (ADM), domain-driven design, and continuous integration—tailored for financial-critical systems. MFLM minimizes temporary adapters, ensures regulatory alignment (e.g., Basel standards), and supports incremental modernization with live integration. A core banking case simulation demonstrates reduced risk, minimal throwaway code, and maintained compliance. Results suggest MFLM enables faster, safer legacy renewal. Future work targets automation of architecture extraction and enhanced data migration strategies.
*Keywords* — Legacy modernization, Modular first legacy modernization, MFLM

## I. INTRODUCTION

Banks worldwide continue to rely on aging mainframe systems—often over 30 years old and written in COBOL—for critical operations like transaction processing and regulatory reporting [9]. These systems are costly to maintain, difficult to evolve, and account for over 100 billion lines of legacy code still in active use [5]. Their limitations hinder business agility, elevate costs, and complicate compliance with evolving standards such as Basel III and BCBS 239 [6].

Regulatory pressure has intensified: as of 2023, only 2 of 31 major banks met BCBS 239 requirements, due in part to legacy-induced data silos [6]. Modernization is now essential, but large-scale "big bang" replacements are rarely feasible. Comella-Dorda et al. note that such full cut-over deployments are "too risky to be admissible," and that modernization projects must demonstrate early value [1]. At the same time, incremental approaches must preserve continuous operations and compliance in areas like payments and ledgers.

Interim modernization states often rely on interface "wrappers" to connect new and legacy components, but excessive adapters introduce technical debt and undermine modernization goals [1]. Any transition phase must also uphold stringent regulatory and security standards. Existing methodologies offer partial solutions. SEI's System Analysis and Migration (SAM) guides phased transitions [1]; OMG's Architecture-Driven Modernization (ADM) offers standardized legacy representations for transformation [3]; and techniques like domain-driven design (DDD) and microservices help modularize legacy architectures [4]. This paper introduces the **Modular-First Legacy Modernization (MFLM)** framework—tailored for banking systems. MFLM prioritizes migration of self-contained, business-aligned modules to minimize dependencies and reduce interface overhead. Each increment ensures compliance and value delivery through continuous integration and automated testing [9]. The rest of this paper is organized as follows: Section II surveys related modernization strategies. Section III details the MFLM methodology. Section IV evaluates its application to a simulated core banking system. Section V presents comparative results, and Section VI concludes with directions for future work.

## II. RELATED WORK

Early research in legacy system modernization identified two contrasting strategies: "Cold Turkey" (Big Bang) replacement and the incremental "Chicken Little" approach [9]. The Big Bang strategy involves replacing the entire system in a single deployment, avoiding interim adapters but concentrating risk and delaying benefit realization. In contrast, Brodie and Stonebraker's "Chicken Little" method advocates gradual migration via gateways, enabling new components to coexist and interact with legacy systems through adapters [8], [2]. Each legacy module is incrementally re-engineered, tested, and integrated, reducing outage risk. While this method handles deployment risk effectively, it provides limited guidance on the prioritization or grouping of components for migration—decisions that significantly impact adapter proliferation and developer effort [8].

To address this, the Software Engineering Institute (SEI) proposed the System Analysis and Migration (SAM) approach [1], which applies structural analysis to optimize incremental migration. SAM aims to minimize scaffolding code, group functionally cohesive elements, and balance effort across iterations. In a case study on a 2-million-line COBOL system,

Comella-Dorda *et al.* [1] used call graphs to identify modular clusters with low external dependencies, guiding a five-phase migration over six years. The approach prioritized early, low-risk increments while balancing cost, technical effort, and functionality delivered per phase.

Complementing SAM is the Object Management Group's (OMG) Architecture-Driven Modernization (ADM) initiative. ADM supports model-based modernization via standards like the Knowledge Discovery Metamodel (KDM), which captures a platform-independent view of legacy systems [3]. KDM enables analysis of structural dependencies and supports transformations into modern platforms using a model-driven pipeline—from source to Platform-Independent Models (PIM) and then Platform-Specific Models (PSM) [8], [3]. While ADM excels in knowledge extraction and tool interoperability, it is domain-agnostic and leaves architectural partitioning decisions to system architects.

Modern techniques such as Domain-Driven Design (DDD) and microservice architectures extend these foundations. DDD advocates modeling software around bounded business contexts (e.g., Payments, Risk, Customer Data), promoting modularity and business alignment [4]. Applying DDD to modernization enables incrementing along meaningful business domains, with tight stakeholder collaboration and clear boundaries. Industry standards like the Banking Industry Architecture Network (BIAN) offer reference models to guide domain decomposition [10], [13]. Recent applications of DDD and BIAN templates, including AI-assisted service design, show promise for large-scale banking modernization [10].

Together, these approaches contribute essential capabilities: SAM and Chicken Little for risk-managed incremental delivery [1], [8], ADM for structural comprehension [3], and DDD for aligning technology with business goals [4], [10]. However, adapter minimization and regulatory compliance—critical in banking—remain under-emphasized. This paper presents the **Modular-First Legacy Modernization (MFLM)** framework, which synthesizes these strategies. MFLM delivers domain-aligned modernization increments, explicitly optimized for minimizing integration overhead and ensuring compliance at every phase. The next section details MFLM's step-by-step methodology.

## III. METHODOLOGY

Overview: The Modular-First Legacy Modernization (MFLM) framework is a structured methodology for incrementally modernizing a legacy banking system. It consists of six major phases: (1) Legacy Element Analysis, (2) Business Domain Modeling, (3) Microservice Interface Design, (4) Increment Mapping via Migration Matrix, (5) Adapter Minimization & Validation Testing, and (6) Deployment & Monitoring. These phases are applied iteratively for each modernization increment. Figure 1 illustrates the overall process flow of MFLM.

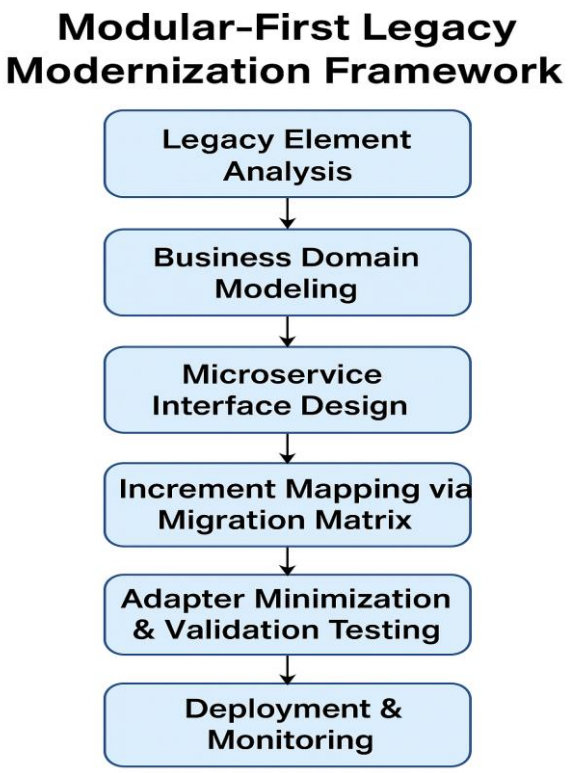


*Figure 1: Modular-First Legacy Modernization Framework – in each cycle, legacy components are analyzed and mapped to business-aligned modules, which are then designed, implemented, and integrated with minimal adapters before deployment.*

### *A. Legacy Element Analysis*

Modernization begins with a comprehensive analysis of the legacy system's structure, dependencies, and execution characteristics. In banking environments, this involves parsing COBOL source code, job control flows, and database schemas; consulting system documentation; and interviewing experienced developers. To formalize this analysis, tools based on OMG's Architecture-Driven Modernization (ADM) standards—particularly the Knowledge Discovery Metamodel (KDM)—are employed to extract a machine-readable model of system elements and their relationships [3]. This yields an inventory of programs, data files, interfaces, and their interconnections (e.g., call graphs, data flows), enabling visualization through dependency matrices. This phase identifies tightly coupled vs. isolated components, maps legacy routines to business functions, and detects domain-aligned clusters (e.g., modules exclusive to "Payments" or "Customer Info"). In the SEI case study, constructing a call graph of a large COBOL system revealed low-dependency clusters that could be incrementally migrated with minimal disruption [1]. Additionally, the analysis documents external interfaces and regulatory integration points—such as payment gateways, reporting systems, Basel risk report generators, and audit logs—ensuring compliance is preserved throughout migration.

### *B. Business Domain Modeling*

In parallel with technical analysis, MFLM incorporates business domain modeling to align system transformation with organizational capabilities. Using Domain-Driven Design (DDD) principles, legacy functionality is mapped to major business domains such as Customer Management, Accounts & Ledgers, Payments Processing, and Risk & Compliance Reporting. Each legacy component identified in Phase 1 is assigned to one or more of these domains. Though monolithic, legacy systems often reveal implicit separations—by subsystem or data structure—that correspond to distinct business functions. These are formalized into *bounded contexts*, each becoming a modular unit in the target architecture. Collaboration with business stakeholders ensures domain boundaries are meaningful and accurate. The output is a **Business Architecture Blueprint**, represented via context maps or component diagrams. Leveraging the Banking Industry Architecture Network (BIAN) reference models [10], [13], this blueprint ensures coverage of critical banking domains (e.g., Payments Execution, Customer Offerings, Risk Analysis) and traces them to compliance responsibilities—such as Basel III reporting under the Risk domain.

### *C. Microservice Interface Design*

With domain-bounded modules defined, the phase focuses on designing their interfaces and interactions—effectively shaping the target system architecture. MFLM adopts a microservices or service-oriented design, where each domain is implemented as one or more services with well-defined APIs (e.g., REST/JSON or gRPC). For each domain module, we define:

- The external interface (e.g., APIs for the Payments Service)
- A refactored data model (inspired by, but not identical to legacy schemas)
- Communication patterns with other services.

Crucially, the design accounts for both the final state (fully modernized modules) and transitional states where legacy and modern components coexist. Adapter interfaces are planned for these hybrid scenarios. For instance, if Payments is modernized before Accounts, an adapter allows Payments to update balances in the legacy Accounts system (e.g., via a temporary database connector or COBOL call). Conversely, if Accounts is modernized first, legacy payments modules call the new service via an adapter. These adapters are kept minimal and one-directional (e.g., proxy or facade patterns), similar to the "wrapper" approach seen in legacy modernization [1]. To further reduce adapter overhead, modules with heavy interdependencies are scheduled for modernization in the same increment. This phase ensures all integration points—between legacy and modern services—are explicitly defined. Compliance, security, and data integrity are also addressed here. For example, if a new service adds validation, not present in legacy systems, the adapter layer ensures regulatory requirements remain satisfied throughout the transition.

### *D. Increment Mapping via Migration Matrix*

With domain modules and their interactions defined, the next step involves assigning modules to modernization increments. This is documented using a **migration matrix**, where rows represent legacy components and columns represent planned increments.

Table 1: Example Modernization Plan – Mapping Legacy Modules to Increments

| Legacy Module | Business Domain | Planned Increment |
|---|---|---|
| Customer Information Management | Customer & KYC Domain | 1 (Phase 1) |
| Account Ledger & Balances | Accounts Domain | 3 (Phase 3) |
| Payments Processing Engine | Payments Domain | 2 (Phase 2) |
| Payment Network Interface | Payments Domain | 2 (Phase 2) |
| Risk Analytics & Reporting | Risk/Compliance Domain | 3 (Phase 3) |
| Audit & Logging subsystem | Compliance Domain | 1 (Phase 1) |

*In this example, Increment 1 modernizes Customer Info and Audit subsystems (which are relatively isolated), Increment 2 tackles the Payments engine and its external interface together (a cohesive pair), and Increment 3 upgrades the Account ledger and Risk reporting last.*

With domain modules and their interactions defined, the next step involves assigning modules to modernization increments. This is documented using a migration matrix, where rows represent legacy components and columns represent planned increments. Following a modular-first strategy, entire modules or tightly coupled groups are migrated together to minimize interim adapter requirements. For example, if the Payments Processing Engine and Network Interface are tightly integrated, both are modernized in the same increment, avoiding the need for mutual adapters. Meanwhile, modules like Account Ledger may be deferred (e.g., to Increment 3), accepting temporary adapters from Payments to legacy systems as a trade-off. As SAM methodology suggests, accepting a few well-placed adapters often yields clearer, domain-focused increments [1]. Trade-offs are evaluated systematically automated tools like SEI's SAM can generate and compare alternative groupings based on metrics such as adapter count and functional cohesion [1].

The selected migration matrix reflects:

- The scope of each increment.
- Adapter connections required per phase.

Increment sizing is aligned with organizational constraints such as budget cycles and staffing. Standalone or low-effort modules may be marked for flexible allocation, allowing them to be inserted opportunistically—an approach successfully used in the SEI case study [1]. The output of this phase is a sequenced, resource-aligned migration roadmap, balancing functional grouping, adapter minimization, and practical delivery constraints.

### E. Adapter Minimization & Validation Testing

Once the scope of an increment is defined, implementation proceeds with development of the new module(s) and required adapters. MFLM prioritizes **adapter minimization**—both in number and complexity. Adapters are treated as temporary scaffolding and kept lightweight, performing only essential tasks such as simple data mappings or bridging calls to legacy APIs [1]. Wherever possible, minor adjustments to legacy components (e.g., output format changes) are used to avoid building new glue code. Each adapter is tracked for **future removal** once both connected modules are modernized. This avoids long-term maintenance of throwaway code and aligns with SAM's principle of reducing non-contributory development effort [1]. This phase also incorporates rigorous **validation testing**: **Module-level tests** verify functionality of the newly implemented service. **Integration tests** ensure correct interaction between new modules and legacy systems via adapters. For example, after implementing the new Payments module (e.g., Increment 2), integration tests verify that payment transactions update legacy account balances and generate accurate legacy reports. If reporting functionality is deferred, stubs may simulate future behavior. Modern **CI/CD pipelines** execute these tests continuously, supporting rapid feedback as components evolve [9]. Compliance validation is also embedded—e.g., Basel III report outputs from the hybrid system are compared with legacy equivalents to catch discrepancies early. Upon successful testing, the increment is deployed in a **staging environment**, running in parallel with legacy components. Final sign-off may include business validation to confirm that new modules produce expected outcomes across critical scenarios.

### F. Deployment and Monitoring

In the final phase of each increment, new components are deployed into production alongside legacy systems using controlled methods such as the gateway/strangler pattern from Chicken Little [8]. Initially, a small subset of transactions is routed to the new service, gradually increasing as confidence builds, or operated in parallel (shadow mode) before full cutover. Gateway transitions are orchestrated to ensure zero downtime by flipping adapters so the new module becomes primary while the legacy becomes secondary [8]. Extensive monitoring, both technical (application performance management, logs) and business (validating transaction totals, balances, and reports), is conducted post-deployment. Anomalies trigger rollback or rapid fixes. New and legacy systems often run parallel for one business cycle to ensure consistency. Upon confirming the new increment meets all requirements, the corresponding legacy components and temporary adapters are retired, thereby reducing the legacy footprint and progressing modernization [1].

Throughout all phases, compliance assurance remains a core principle, especially in banking. Regulatory requirements are integrated into each increment from the outset rather than addressed later. For example, new modules handling customer data are built to comply with GDPR and similar regulations immediately, while risk calculation changes involve validation against Basel standards. Incremental compliance checks prevent audit failures and ensure continuous certification. Modern core banking providers similarly embed regulatory frameworks such as PSD2, GDPR, and Basel III into their systems from inception [7].

In summary, the Modular-First Legacy Modernization (MFLM) methodology combines technical rigor with business alignment. It applies thorough legacy analysis (ADM principles), defines domain-centric targets (DDD), plans optimal migration slices (SAM's incremental approach), and executes with tight feedback loops and governance (CI/CD). The next section presents a simulated case study demonstrating decisions and outcomes across increments.

## IV. CASE SIMULATION: INCREMENTAL MODERNIZATION

To demonstrate the MFLM approach, we simulate modernizing a hypothetical Core Payment System for a mid-sized bank. The legacy system processes customer payments (wires, ACH), updates account balances and generates daily risk reports. It is a typical monolithic COBOL application (~1 million LOC) on an IBM mainframe, supporting batch and online transactions with a relational database. The bank aims to migrate to a cloud-native microservices architecture to improve scalability and comply with real-time payment regulations without disrupting operations. Modernization is executed across three increments using the MFLM framework.

### A. Legacy System Overview

Major components identified during Legacy Analysis include:

- **Payments Engine:** Batch and online payment execution, interfacing with networks (SWIFT, ACH) and ledgers.
- **Account Ledger:** Modules maintaining balances and transaction histories (VSAM/DB2).
- **Customer Info & KYC:** Customer profiles and KYC checks, accessed mainly during payment validation.
- **Risk & Compliance Reporting:** Daily transaction aggregation for risk metrics and regulatory compliance (e.g., BCBS 239 reports).

- **Audit/Logging:** Cross-cutting transaction logging for audits.

In the Business Domain Modeling phase, components are grouped into domains: "Payments" (Payments Engine), "Accounts" (Account Ledger), "Customer" (Customer Info/KYC), and "Risk/Compliance" (Risk Reporting and Audit). Legacy analysis revealed tight coupling between Payments Engine and Account Ledger, while Customer Info was more isolated and Risk Reporting operated independently, influencing migration sequencing.

1) **Increment 1 – Customer and Audit Module Modernization:** The modernization begins with the relatively isolated Customer Info and Audit/Logging modules to minimize risk and validate the tech stack [1]. New Customer and Audit services are implemented as microservices with independent databases. Integration with the legacy Payments Engine is achieved using two adapters: a Customer Lookup Adapter (replacing direct file reads with API calls) and an Audit Log Adapter (sending messages via middleware queues). Testing includes parallel runs to ensure customer checks and audit logs match legacy behavior. Post-deployment, both legacy and new services operate in parallel before full cutover. Metrics show a 20% improvement in customer data response times without compliance issues. After a successful transition, the old customer database and audit logs are decommissioned, removing associated adapters, achieving ~20% system modernization.

2) **Increment 2 – Payments Engine and Network Interface Modernization:** This increment targets the core Payments Engine and external network interfaces. A new Payments Microservice and Payments Gateway Service are developed. As the Account Ledger remains legacy, an Account Update Adapter is created for balance updates, while a Legacy Payments Adapter ensures backward compatibility for unmodernized components. Direct integration with the previously modernized Customer Service simplifies data retrieval. Extensive parallel testing verifies financial accuracy, transaction integrity, and compliance. Canary releases gradually shift transaction loads to the new system, ensuring tight coordination between new and legacy database writes. Monitoring shows improved scalability and a 15% reduction in payment failures. Regulators are kept informed throughout. Upon completion, ~70% of system functionality is modernized, covering Payments, Customer, and Audit modules.

3) **Increment 3 – Account Ledger and Risk Reporting Modernization:** The final increment modernizes the Account Ledger and Risk/Compliance reporting modules, which are interdependent. A new Accounts Service is developed with data migrated to a distributed database, while a new Reporting & Analytics module is implemented to compute risk metrics using modern services. Dual writes between legacy and new databases temporarily maintain consistency during cut-over. The Payments service configuration is updated to interact directly with the new Accounts service. Risk reporting is transitioned to the new data sources after parallel validation runs ensure output consistency, including scrutiny of rounding and historical comparison.

Integration tests and risk officer sign-offs precede production deployment, scheduled during a maintenance window. Post-migration, all systems access the new Accounts service, legacy systems are archived, and infrastructure costs are reduced. By the end of Increment 3, all core services (Customer, Payments, Accounts, Audit, Risk Reporting) are modernized. Modularization enables future extensions, and incremental delivery maintains stakeholder support across 18–24 months.

**B. Key Outcomes:**

Five adapters were implemented (Customer lookup, Audit log, Account update, Legacy-to-new payments call, Dual-write), most decommissioned quickly, validating the modular migration strategy [1]. No unplanned downtime occurred; gradual switchovers maintained continuous operation. Regulatory compliance was ensured through parallel reporting and early risk officer involvement, aligning with industry modernization practices [7]. The overall project spanned 2.5 years with incremental releases every ~9 months, allowing phased funding and scope adjustment [1]. Developer productivity improved over time as pipelines and infrastructure matured. The new system achieved ~25% higher transaction throughput, enabled modular scaling, and supported dynamic expansion of services.

This simulation illustrates the effective application of MFLM to a core banking system, highlighting how modular incremental modernization reduces risk, improves performance, and strengthens compliance.

Table 1: Summary of modernization Increments in core banking system.

| Criteria | Big Bang Rewrite | Traditional Incremental (ad-hoc) | Architecture-Driven (ADM-based) | MFLM (Proposed) |
|---|---|---|---|---|
| **Operational Risk** | Very high – long outage or cut-over risk; full fallback needed if fails [1]. | Moderate – staged, but risk depends on planning quality; potential mid-project failures. | Moderate – emphasizes understanding, can mitigate some risk but doesn't itself execute incrementally. | Low – fully incremental, system remains operational at all times [1]; gateways used to isolate risk per |

| | | | | |
|---|---|---|---|---|
| | | | | increment [8]. |
| **Adapters / Integration** | None during dev (clean slate), but requires huge integration at cut-over. | Potentially many ad-hoc adapters if increments not well-scoped; danger of "spaghetti" interim architecture. | Focus on model extraction; integration issues discovered later when implementing changes. | Minimal adapters – planned and optimized grouping yields fewer, simpler adapters [1], systematically removed as increments progress. |
| **Development Effort** | Very high upfront; duplication of legacy functionality from scratch. | Incremental effort but can be inefficient if redundancies or rework occur due to poor planning. | High analysis effort; actual redevelopment may still be large; risk of analysis paralysis. | Incremental effort focused per domain; less rework due to cohesive module scope; first increment overhead amortized over later ones. |
| **Time to First Value** | Long (years) – no user-visible benefit until completion. | Faster than big bang if increments deliver something, but without clear domain focus, early increments might deliver trivial improvements. | Long – initial phase focuses on documentation /modeling, which may not yield running code for some time. | Short – initial increment delivers functional improvement within months; each subsequent increment provides new capabilities or performance gains. Early benefits demonstrated (e.g. better KYC after increment 1) [1]. |
| **Business Alignment** | Risk of mismatch – requirements may drift over long dev cycle; hard for business to envision final product. | Varies – could align if business involved each step, but ad-hoc increments often technically driven. | Strong on documentation, but gap between models and actual business value if not translated to implementation. | Strong – domain-driven approach keeps modernization aligned with business context [5] Business stakeholders engaged per domain |
| | | | | module; feedback incorporated continuously. |
| **Compliance Assurance** | Big risk – new system must be certified at end; any miss can delay go-live. | Medium – increments might overlook compliance in rush to deliver slices, unless explicitly checked. | High understanding of existing compliance mechanisms, but not inherently ensuring new implementation complies until tested. | High – continuous compliance verification each increment. Regulatory reports run in parallel to ensure consistency [6]. No final surprises as each piece is compliant by design. |
| **Flexibility & Adaptability** | N/A until complete; architecture decided at start, hard to change mid-flight. | Medium – can adjust next increments based on feedback, though earlier mistakes costly to fix. | High flexibility in analysis models but translating that to code changes is indirect. | High – plan can be adjusted between increments (e.g., if one domain needs faster rollout due to regulatory deadline, we can reorder). The modular nature means changes in one increment have limited impact on others. |
| **Final Architecture Quality** | Potentially high if done perfectly (clean-sheet design). But risk of under/over-engineering since no intermediate validation. | Could be suboptimal if increments were not planned with whole architecture in mind (architecture may just evolve, not designed). | High theoretical quality (due to thorough architecture extraction) but depends on execution of refactoring. | High – intentionally designed target architecture (domain microservices) implemented stepwise. Each step is validated, so architecture is proven to work. Domain boundaries in final |

| | | | | |
|---|---|---|---|---|
| | | | | system reflect real business needs (thanks to DDD). |

From the above comparison, MFLM achieves a balanced performance across the criteria, whereas other approaches excel in some dimensions but fall short in others. Notably:

- **Risk Mitigation:** MFLM minimizes operational risk by maintaining continuous system availability. In the case simulation, no downtime occurred, contrasting with the concentrated risk of a Big Bang approach. Even incremental strategies can fail without careful coordination; MFLM localizes failures to specific modules, enabling rollback without system-wide disruption. This aligns with the Chicken Little philosophy of progressive deployment [8] but with enhanced increment planning.

- **Adapter Overhead:** Minimizing adapters was a key objective. Adapters, though necessary for interoperability during migration, introduce complexity [1]. MFLM required only five adapters versus an estimated eight to ten with uncoordinated migration. This validates the SAM approach, where fewer adapters reduce development effort, risk, and maintenance [1]. Unlike ADM, which maps dependencies without prescribing sequencing, MFLM explicitly plans to minimize interface work.

- **Timeline and Value Delivery:** MFLM delivers value early and regularly, crucial for maintaining funding in banking environments. Incremental improvements, such as faster KYC processing (Increment 1) and increased payment throughput (Increment 2), align with agile principles and stakeholder expectations [4]. Big Bang rewrites, by contrast, often fail due to a lack of interim deliverables.

- **Business and Regulatory Alignment:** Domain-driven planning ensures alignment with business and regulatory needs. Engaging business units in each phase allowed real-time adjustments, such as incorporating new AML checks during Increment 2. Compliance was validated incrementally, consistent with guidance advocating iterative modernization to manage risk [4], [5].

- **Architecture and Quality:** MFLM produced a modular, microservices-based architecture validated incrementally in production. This evolutionary model, similar to the strangler pattern [10], reduces architectural risks compared to greenfield rewrites. While ADM provides static "as-is" and "to-be" models, MFLM ensures real-world validation through phased deployment.

- **Cost and Effort:** MFLM spreads modernization costs across increments, enabling funding flexibility and risk-managed progress. Although incremental execution introduces overhead (e.g., adapters, parallel operations), minimizing scaffolding aligns with SAM's findings [1]. Compared to ADM-heavy approaches that emphasize upfront modeling, MFLM maintains a balance between planning and execution to ensure steady modernization without excessive cost or delay.

## V. CONCLUSION AND FUTURE WORK

MFLM offers a holistic modernization strategy balancing technical progress with business continuity and compliance. It mitigates the risks of Big Bang rewrites by segmenting modernization safely, and improves upon naive incrementalism through architecture-driven, domain-focused planning. Case simulation results (e.g., adapter count, continuous uptime) validate its effectiveness, operationalizing the best practices of phased execution with structured planning [4]. However, MFLM introduces trade-offs. Significant upfront analysis is required, and poor legacy documentation can complicate planning, though ADM techniques and reverse engineering mitigate this risk. Operating old and new systems in parallel introduces synchronization complexity, managed through careful design. Although a flawless Big Bang could be faster, such outcomes are rare in banking IT. Human factors also impact success; incremental modernization demands consistent vision, patient execution, and strong executive sponsorship to prevent mid-course resource diversion.

In conclusion, MFLM provides a balanced, low-risk modernization pathway, blending continuous improvement with strategic alignment. It presents a viable alternative for banks that have encountered failures with Big Bang strategies or prolonged analysis efforts. The following section concludes the paper and outlines areas for future work.

Future Work, while the MFLM approach proved effective in our analysis, there are several areas of future work and potential enhancements:

- **Tool and Automation:** Automating parts of MFLM can further reduce effort and risk. Advanced program analysis, AI-driven code comprehension, and machine learning models for legacy languages (e.g., COBOL) could enhance Legacy Element Analysis and architecture extraction. Automation could also aid adapter generation and CI pipeline setup. Extending ADM models (e.g., KDM) with AI for migration path recommendations presents a promising research avenue, supporting continuous modernization.
- **Data Migration and the Butterfly Method:** Data migration remains a major challenge. The Butterfly methodology, which progressively transforms legacy databases without gateways [11], [12], could enhance MFLM by enabling continuous replication and schema

evolution, particularly in critical areas like Account Ledger migration.

- **Quantitative Optimization Models:** Increment planning was largely qualitative. Future work could formalize this as an optimization problem, using integer programming or heuristics to partition legacy components, minimizing adapters and balancing risk. Extending SAM's early heuristics [1] with modern solvers and richer ADM data could yield near-optimal migration strategies.

- **Continuous Integration of Compliance:** Embedding compliance validation into CI/CD pipelines could institutionalize regulatory checks. Automating Basel III report generation, audit verification, and security compliance in each cycle would strengthen risk management, benefiting heavily regulated industries beyond banking.
- **Generalization and Case Studies:** Applying MFLM to other legacy domains (e.g., insurance, government systems) would test its broader applicability. Real-world case studies tracking metrics like defects, downtime, and adapter volume would provide empirical validation and strengthen industry adoption.
- **Legacy Workforce Transition:** Modernization also requires addressing workforce shifts. Strategies like event storming during Legacy Analysis and mentorship pairing between veteran and modern engineers could preserve critical domain knowledge during transitions.

Legacy modernization is a complex but navigable journey. MFLM offers a structured, value-driven, and compliant path by aligning technical analysis, business needs, and risk mitigation. As demonstrated, modular incremental modernization avoids the risks of Big Bang failures while delivering agile, future-ready banking cores. Continued collaboration between practitioners and researchers is encouraged to refine and extend this framework.

## REFERENCES


[1] S. Comella-Dorda, G. Lewis, P. Place, D. Plakosh, & R. Seacord. "Incremental Modernization for Legacy Systems." CMU/SEI-2001-TN-006, Software Engineering Institute, 2001. Case study and methodology for iterative COBOL system migration.

[2] M. L. Brodie & M. Stonebraker. Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach. Morgan Kaufmann Publishers, 1995. Classic text introducing the "Chicken Little" incremental migration strategy using gateways.

[3] Object Management Group (OMG). "Architecture-Driven Modernization (ADM) Standards." OMG ADM Task Force, 2003–2017. Defines models like the Knowledge Discovery Metamodel (KDM) for representing existing software to enable modernization.

[4] Coherent Market Insights. "How to Expedite the Modernization of Legacy Banking System." CMI Blog, 2023. Highlights the use of Domain-Driven Design for safe, incremental legacy bank system re-architecting (bounded contexts to reduce entanglement).

[5] C. Chowdhury. "How Domain-Driven Design Can Boost Legacy Systems Modernization." Cognizant 20-20 Insights, Oct. 2020. Discusses aligning legacy modernization with business domains and ubiquitous language so that IT and business move in lockstep.

[6] Informatica. "Addressing the Challenges of BCBS 239 Compliance with a Modern Approach to Data Management." Informatica Blog, Mar. 18, 2025. Explains how legacy IT silos impede risk data aggregation and why modern data integration is essential for Basel compliance.

[7] Basel Committee on Banking Supervision. "Basel III framework – regulatory standards for banks." BIS, 2017. (Referenced via Basikon article) Modern core banking solutions incorporate strict regulatory requirements (PSD2, GDPR, Basel III) by design.

[8] S. Goos. "Model-Driven Legacy System Modernization." MSc Thesis, University of Twente, 2014. (Used for reference on Chicken Little and ADM approaches).

[9] Fraunhofer IESE. "Legacy Systems / System Modernization." Fraunhofer Institute webpage, 2021. Describes legacy modernization challenges in banking (mainframe costs, need for CI/CD, etc.) and mentions strategies (Big Bang, Chicken Little, Butterfly).

[10] Alfredo Muñoz. "Industrialize the modernization of banking systems using BIAN standard." Medium Blog, Oct. 25, 2024. Presents an approach combining BIAN, DDD, and automation, reinforcing the benefit of standardized service domains in bank system modernization.

[11] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wade, R. Richardson and D. 0' Sullivan, 'Migrating Legacy Systems: From a Caterpillar to a Butterfly', Trinity College Technical Report, January 1997.

[12] A, Sivagnana Ganesan, T. Chithralekha. Comparative Review of Migration of Legacy Systems. Pondicherry University Puducherry.

[13] Hans Tesselaar, Klaas de Groot, Guy Rackham. A framework for the financial services industry.