International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

# Intelligent Mainframe Modernization: Applying Domain-Driven Design to Extract Business Capabilities from Legacy Systems

**Kalyana Sundaram Chidambaram**

**Abstract:**

**Background**: Legacy mainframe systems are a large part of an enterprise IT ecosystem today. Their tightly coupled architecture makes them difficult to modernize, and customary approaches such as re-architecting or lift-and-shift migration present operational and functional risk due to the undocumented business logic that has spread through a codebase over decades.

**Objective**: To present a framework that combines domain-driven design (DDD) and reverse engineering techniques to support the business-oriented, systematic modernization of legacy mainframe systems.

**Methods**: Bounded context analysis identifies business capabilities from monolithic mainframe architectures. Static and dynamic analysis recognize areas in a system that are tightly coupled and potential candidates for service boundaries. These boundaries are then implemented as independent microservices with RESTful APIs, deployed to container-based cloud-native infrastructures with automated testing and validation pipelines.

**Results**: The framework leads to the improvement of system modularity through domain-oriented decomposition of services, reduced structural coupling through API-level interaction, improved reliability through cloud failover, and reduced defect density and operational complexity through testing and deployment automation. By eliminating intermediate messaging layers, it increases event processing efficiency and the system's topological simplicity.

**Conclusions**: The domain-driven modernization can be a scalable incremental alternative to risky wholesale replacement strategies through encapsulating business logic within a technical architecture conforming to the natural structure of the business domain, lowering the risk of transformation or easing the incremental technical evolution of enterprise systems.

***Keywords:*** *Mainframe Modernization, Domain-Driven Design, Microservices, Bounded Contexts, Reverse Engineering, Restful API*

**Practitioner Points**

- Domain-driven design is a pattern for incremental low-risk mainframe modernization that focuses on domains of business capability rather than technical components.
- Using a RESTful API instead of message queuing reduces processing costs and makes enterprise integration easier.
- The modernization cycle is completed with the cloud-native deployment, automated testing, and failover across multiple regions with near-zero downtime.

*Independent Researcher, USA*

## 1. Introduction

Enterprise mainframe systems have supported mission-critical business operations across industries with high transactional rates, stringent requirements for data consistency, and demands for operational

reliability. Built several decades ago using procedural programming languages, these systems represent complex business logic in architectures which were not designed for modularity or independence. deployment. As companies demand digital transformation at an accelerated pace, many are finding these architectures to be inflexible planned liabilities [1].

The most common modernization strategies, full system rewrites using one of the idioms of the language, language-level translation, and full infrastructure migration, build on each other, exacerbating these risks. Rewrites require complete reconstruction of implicit business logic (rarely documented) and almost inevitably result in behavioral regression or functional loss. Lift-and-shift migration strategies allow for modernization of the cloud infrastructure but do not overcome the limitations of the existing architecture. Neither strategy provides a systematic way to retain business semantics from legacy code while introducing new architectural properties [2].

A more principled approach is the intersection between domain-driven design (DDD) and reverse engineering: DDD is an approach to software architecture where the business domain is the main organizing principle for the structure of a software system and where a bounded context defines a logical boundary within which a certain domain model is defined [3]. The behavior of a legacy system can be reverse engineered from the source artifacts to derive the implicit domain structure [4].

These approaches change the way we think about the modernization problem, from 'How do we replace our legacy system?' to 'How do we extract and re-express the business capabilities encoded in our legacy system as a modern, modular, and evolvable architecture?' This has been confirmed by other modernization situations; related tertiary studies that survey the literature in the microservices modernization space find domain-aligned decomposition to produce better modularity, coupling, and deployability results than technical decomposition approaches [5]. Studies of real-world microservices design with practitioners report that DDD concepts like bounded contexts, aggregates, and domain events are the primary inputs to effective service boundary design [6].

The paper describes an end-to-end approach to applying the principles of DDD to the modernization of mainframe systems from domain identification through system decomposition, microservices design, API and event-driven system integration, and cloud-native system operationalization. Each phase of the modernization approach is based on the existing body of literature on DDD, legacy modernization, and microservices. The remainder of the article is structured as follows: methodology (Section 2), results of each phase of the framework (Section 3), calculated outcomes (Section 4), and conclusions (Section 5).

## 2. Methodology

### 2.1 Framework Design Rationale

The proposed framework is a staged, domain theory-driven transformation pipeline based on three methodological principles from DDD and legacy modernization literature.

First, however, is the issue of business semantics. The business capability of legacy systems is encoded in a way that technical decomposition strategies (such as metrics, call graphs, and data flow analysis) would not have otherwise discovered. DDD offers a shared vocabulary in business terms to specify the system architecture, where the knowledge of the domain is captured [7].

Second, decomposition should be stepwise with operational safety, whereas one large-scale transition concentrates all risk in the single transformation. An incremental approach, where bounded contexts are extracted, and the legacy system is refactored in stages, reduces risk, as the legacy system is kept running during the process via sequential bounded-scope transformations [4].

A third is that the output of the transformation should be architecturally modern, meaning that it should be independently deployable, API-accessible, scalable in a cloud-native way, and continuously deliverable to deliver the organizational agility that is the objective of modernization [5].

### 2.2 Domain Identification Through Static and Dynamic Analysis

The first phase of the framework involves static and dynamic analysis to produce the domain model of the

legacy system. Static analysis of program listings, copybooks, data definition files, and job control language procedures is used to create a dependency graph with program call chains, shared data access patterns, and module interdependencies [8]. Dynamic analysis tools instrument the application to gather information about runtime behavior, such as the execution path taken, transaction frequency, and data flow.

Taxonomy-based service identification approaches categorize these service identification techniques according to dimensions, such as the input artifacts, decomposition criteria, and granularity. These dimensions can be used to select and combine analysis techniques for a specific legacy system [9]. Domain metric-driven approaches turn DDD concepts into metrics, such as the metrics for intra-context cohesion and inter-context coupling, and use them to assess and evolve the identified candidates [10].

The primary outcome of this phase is the service candidate map, which is an organized representation of the candidate bounded contexts, their domain responsibilities, their data ownership, and their relationships with other bounded contexts. It is the main input to the design phase and a reference for communicating the modernization plan to its technical and business stakeholders.

| Analysis Type | Input Artifacts | Output | Primary Purpose |
|---|---|---|---|
| Static Analysis | Source code, copybooks, JCL procedures, data definition files | Program dependency graph; shared data access patterns | Identify structural coupling and module interdependencies |
| Dynamic Analysis | Runtime execution traces, transaction logs, workload profiles | Execution path map; transaction frequency distribution | Capture behavioral groupings under realistic conditions |
| Domain Metric Evaluation | Candidate service clusters from static/dynamic outputs | Cohesion and coupling scores per candidate context | Evaluate and refine decomposition quality |
| Taxonomy-Based Classification | Decomposition method inputs and criteria | Structured selection of appropriate analysis techniques | Match analysis approach to legacy system characteristics |

TABLE 1: Analysis Techniques Applied in Domain Identification Phase [9, 10]

### 2.3 Microservices Architecture and the Anti-Corruption Layer Strategy

Each bounded context is implemented as a separate microservice with its own domain model and API. The design of each microservice seeks to preserve the business semantics of the legacy functionality it replaces in a form that uses the structural properties favored by a good microservices architecture [6].

During the overlap period where both the new and old versions of the services have to coexist, the anti-corruption layer pattern allows a clear boundary between the new service's domain model and the legacy system's data model and semantics [3]. This is important in a mainframe context, where data formats, character encodings, and field semantics often differ widely between the legacy and new systems.

The model-driven service identification approaches formalize the service design process by providing metamodels and transformation rules that are uniformly applied on the service candidates, which reduces the variability and the probability of human error that exists in manual service design methods [11].

| Design Property | Description | Anti-Corruption Layer Required? |
|---|---|---|
| Data Ownership | Each bounded context owns its data; no shared dataset access across contexts | Yes—during transition, legacy shared datasets require translation |
| API Contract Stability | The service exposes capabilities through versioned, stable API endpoints | Partially—contract must absorb legacy semantic differences |
| Domain Model Isolation | The internal model uses the ubiquitous language of the bounded context | Yes—legacy data structures must be translated at the boundary |
| Independent Deployability | Service can be deployed, scaled, and updated without coordinating with other services | No—resolved once the bounded context is fully extracted |
| Semantic Translation | Differences in field semantics, encoding, and formats between legacy and modern representations | Yes—until full extraction is complete |

TABLE 3 — Bounded Context Design Properties and Anti-Corruption Layer Applicability [3, 4]

### 2.4 API-Driven Integration and Cloud-Native Deployments

Modernized services communicate over RESTful APIs, using JSON as the data format. The model replaces legacy middleware and inter-process communication technologies (such as message queues, proprietary integration middleware, legacy data formats, and communication protocols) with modern direct service-to-service communication via lightweight APIs [6]. Standardized API contracts with interface specifications enable a stable interface for downstream consumers and allow new applications to be added.

Cloud-native deployment provides the operational substrate for the modernized architecture, allowing for container-based isolation, orchestrated deployment, and declarative configuration to independently scale and deploy individual services [12]. Multi-region deployment and automatic failover are used to achieve the same availability characteristics of the incumbent mainframe system within the modernized architecture. Continuous integration and automated testing pipelines provide the delivery infrastructure to enable a multi-service architecture to operate at an enterprise scale [8].

## 3. Results

### 3.1 Domain Decomposition Outcomes

Applying this decomposition mechanism to legacy mainframe applications (e.g., COBOL batch processing, VSAM data access, and transaction processing) yields a domain map, showing a set of functionally coupled programs and/or data sets that can be treated as bounded contexts (with clear responsibilities and boundaries). Static analysis of the program interaction graph yields clusters of programs that co-access the same data sets and call the same utility modules. Dynamic analysis identifies the same clusters by finding groups of transactions that correspond to coherent business operations.

Decomposition's core understanding is that many large lumps of legacy code that have structurally similar undifferentiated procedural logic implement different business capabilities, often because the original legacy development methodology did not create explicit modular boundaries. Decomposition makes these implicit boundaries explicit and derives an explicit decomposition that maps to real-world business processes, not technical ones [7].

Semi-automated approaches for migrating legacy architectures to a modern architecture, using static analysis to inform and drive the identification of service boundaries, have been shown to be more efficient at manual domain mapping and serve better to reduce inconsistencies than a fully expert-driven

approach when using structured decomposition techniques [4].

### 3.2 Microservices extraction to reduce coupling

To turn the identified bounded contexts into microservices requires also changing the systems' coupling. In existing systems, any number of programs understanding the same model are coupled, so a change in one program ripples through the system. However, in the new architecture, each service has its own data store and only exposes the data to other services through an API [5].

Practitioner studies indicate the alignment of microservices API design with DDD constructs is a consistent predictor of coupling reduction. When a service's boundary aligns with a domain's boundary, the dependency between two services is semantic (responsible service and client share an API contract) rather than structural. This means that changes to a service's domain boundaries do not affect consumers (as long as the service's API contract is honored) [6].

Another coupling reduction benefit of the architecture is the removal of business rules (business logic) that were naively redundantly implemented in multiple programs in the legacy system. These duplicated rules can be consolidated into shared services, providing stable APIs without the inconsistency caused by different implementations of the same rule in each legacy program [13].

### 3.3 API Performance and Integration Improvements

Replacing the message queue communication with a direct RESTful API can lead to performance improvements since there is some overhead in the enqueue/dequeue process and in converting between message formats and handling dead letters. Communicating directly with the API eliminates intermediate steps and reduces the number of processes involved in returning a transaction path [6].

JSON, a low-overhead, ubiquitous serialization format for exchanging data over an API, has less serialization overhead than the binary or flat-file records commonly used for mainframe data interchange. The standardized API contract makes it faster to onboard new consumer applications, with a common interface for all consumers that does not require a custom format translation layer [14].

The reference architectures for domain-driven, microservices-based systems in high-reliability enterprise contexts verify the feasibility of API-driven integration and the simplicity of integration provided by APIs, which increases with the number of applications dependent on the APIs [12].

| Attribute | Message Queue-Based Communication | RESTful API with JSON |
|---|---|---|
| Processing Steps per Transaction | Multiple: enqueue, transform, dequeue, process | Minimal — direct request-response |
| Data Format | Binary or fixed-format records | Lightweight JSON |
| Latency Profile | Higher — asynchronous with queuing delay | Lower—synchronous, direct |
| Failure Points | Queue infrastructure, format converter, dead-letter handler | Service availability only |
| Integration Complexity | High — custom adapters per consumer | Low-standardized contract reused across consumers |
| Operational Overhead | Queue management, schema versioning, and monitoring | Standard API management tooling |
| Scalability of Integration | Linear increase with each new consumer | Stable — consumers share a contract without custom translation |

TABLE 4: Legacy Communication Mechanisms vs. RESTful API Integration [6, 12]

### 3.4 Operational Reliability Through Cloud-Native Deployment

Such automated health checking, process restart on failure, and load balancing between replicated services are not supported natively in legacy mainframe deployments or require proprietary hardware in some systems to achieve similar results. Multi-region deployment allows automatic failover of traffic whenever an instance in a region becomes unavailable, ensuring service continuity [10].

Automated unit testing on each commit, automated integration testing, and automated contract testing can catch defects before they reach production and lower the risk of regressions. Contract testing is particularly relevant for microservices where a behavioral change can propagate to consumers by breaking the API contract between the producer and consumer microservice. Automated contract testing can be a mechanism to prevent such a failure [8].

Additionally, fewer systems to operate mean the operations topology becomes simpler, or the number of independently deployable and monitored components goes down. This typically reduces the operational surface area, the chance of system failure caused by the interaction of components, and the workload on operations staff taking care of a production environment [9].

| Cloud-Native Capability | Mechanism | Modernization Benefit |
|---|---|---|
| Container-Based Isolation | Each service runs in an independent container with its own runtime | Eliminates shared library conflicts; enables independent deployment cycles |
| Multi-Region Deployment | Active-active or active-passive regional replication | Continuous operation during infrastructure failure; near-zero downtime transition |
| Automatic Failover | The orchestrator reroutes traffic from failed instances to healthy replicas | Matches or exceeds mainframe hardware availability characteristics |
| Independent Scaling | Per-service resource allocation based on demand | Efficient compute utilization eliminates coarse-grained capacity provisioning |
| Continuous Integration Pipeline | Automated build, test, and deployment on code commit | Early defect detection; consistent releases; reduced regression risk |
| Contract Testing | Automated validation of API response conformance to consumer expectations | Prevents behavioral regressions from propagating across service boundaries |

TABLE 5 — Cloud-Native Deployment Capabilities and Modernization Benefits [8, 9, 10, 12]

## 4. Discussion

### 4.1 Domain-Driven Design as a Modernization Enabler

The findings support the claim that DDD provides a materially better basis for legacy mainframe modernization than technical decomposition methods do. The key difference is that DDD uses business semantics as its primary means for decomposing systems. Services resulting from DDD are stable with respect to business change, aligned with the organizational structure, and understandable to technical and business people alike [7].

At a high level, a system whose long-term architecture is organized around the structure of the business domain is naturally better positioned to accommodate change as business needs evolve. Change is scoped down to the business domain where change is needed. This property, sometimes called "evolvability" (again,

depending on context), is the main benefit of domain-driven modernization in the long term compared to other approaches that may seem to technically modernize systems but do not correct the root cause for the structural mismatch between systems and underlying businesses [3].

Other development models that share DDD's focus on functional boundaries, such as feature-driven development in decision support system development, have found that aligning teams with functional boundaries increases both speed of development and team autonomy [15].

### 4.2 Limitations and Considerations

The applicability of this approach depends on several factors. Firstly, a precondition for effective domain identification is that there are domain experts who can provide sufficient knowledge of the business context of the legacy system. In the absence of institutional knowledge, reconstruction of the implicit domain model from source artifacts is much more difficult. The anti-corruption layer pattern is useful during an incremental transition, giving rise to temporary architectural complexity that should be managed and ultimately removed as modernization progresses.

Distributed transactions in the microservices environment also make service boundaries more challenging to set. It can be costly to coordinate frequent transactions across service boundaries, and this can reverse the benefits of decomposition. However, analyzing the ownership of data during domain identification is not guaranteed to eliminate this problem within systems that have a large number of transactional dependencies [5].

## Conclusion

This article has described a method for applying domain-driven design for modernization projects that aim to transform legacy mainframe systems. Using business semantics for decomposition, a modernized architecture can adhere to the structure of the organizational domain. Written using cloud-native principles, it is designed to be resilient to technical failure and can be continuously evolved via automated delivery pipelines.

The stages of the framework are domain identification, microservices modeling, API integration, cloud-native operationalization, and incremental modernization. They allow a manageable reduction of transformation risk through multiple bounded stages without jeopardizing the operational stability of the enterprise. A structural design pattern for managing the coexistence of legacy and modernized code is the anti-corruption layer pattern.

We find that domain-driven modernization produces better modularity, decoupling, performance, and operability than technical modernization because it addresses the root cause of the problem faced by the legacy system, which is that the technical architecture does not explicitly match the business domain. In future work, we can explore the use of automated domain identification tooling to reduce the analysis effort on the decomposition step , and the use of AI-based code analysis tooling to discover implicit domain boundaries in large-scale legacy systems.

## References

[1] Duc Minh Le et al., "Domain-driven design patterns: A metadata-based approach," 2016 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2016, doi: 10.1109/RIVF.2016.7800302. [Online]. Available: https://ieeexplore.ieee.org/document/7800302

[2] Lucas Fernando Fávero et al., "A Systematic Mapping Study on the Modernization of Legacy Systems to Microservice Architecture," Appl. Syst. Innov. 2025. [Online]. Available: https://www.mdpi.com/2571-5577/8/4/86

[3] Sam Peng and Ying Hu, "Anticorruption: a domain-driven design approach to more robust integration," OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications companion, 2007. doi: 10.1145/1297846.1297966. [Online]. Available: https://dl.acm.org/doi/10.1145/1297846.1297966

[4] Irina Petrariu et al., "A Comparative Study of Unsupervised Anomaly Detection Algorithms used in a Small and Medium-Sized Enterprise," International

Journal of Advanced Computer Science and Applications (IJACSA), 2022. doi: 10.14569/IJACSA.2022.0130908. [Online]. Available: https://thesai.org/Publications/ViewPaper?Volume=13&Issue=9&Code=IJACSA&SerialNo=108

[5] Daniele Wolfart et al., "Modernizing Legacy Systems with Microservices: A Roadmap," EASE '21: Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering, 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3463274.3463334

[6] Daniele Wolfart et al., "The route optimization and fare setting research of feeder transit system related to urban rail transit," 2021 2nd International Conference on Urban Engineering and Management Science (ICUEMS), 2021. doi: 10.1109/ICSA51549.2021.00011. [Online]. Available: https://ieeexplore.ieee.org/document/9426726

[7] Paulius Danenas and Gintautas Garsva, "Domain Driven Development and Feature Driven Development for Development of Decision Support Systems," Information and Software Technologies, 2012. doi: 10.1007/978-3-642-33308-8_16. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-33308-8_16

[8] Lucas Fávero et al., "Micro4Delphi: A Process for the Modernization of Legacy Systems in Delphi to Microservice Architecture," 27th International Conference on Enterprise Information Systems, 2025. doi: 10.5220/0013365700003929. [Online]. Available: https://www.scitepress.org/PublicationsDetail.aspx?ID=rgjeFdmW8vk=

[9] Hareem Sahar et al., "How are issue reports discussed in Gitter chat rooms?" Journal of Systems and Software, 2021. doi: 10.1016/j.jss.2020.110868. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302429

[10] Sai Thu Ya Aung et al., "Blockchain-Based Implementation for Integration of DNA Profiles Information Systems," 2020 - 5th International Conference on Information Technology (InCIT), 2020, doi: 10.1109/ISSREW51248.2020.00060. [Online]. Available: https://ieeexplore.ieee.org/document/9310775

[11] David Alberto García Arango et al., "Design and validation of a comprehensive evaluation proposal for MOOC-type courses," 2020 15th Iberian Conference on Information Systems and Technologies (CISTI), 2020. doi: 10.23919/CISTI49556.2020.9141096. [Online]. Available: https://ieeexplore.ieee.org/document/9141096

[12] Nicolò Pasini et al., "A virtual suturing task: proof of concept for awareness in autonomous camera motion," 2022 Sixth IEEE International Conference on Robotic Computing (IRC), 2023, doi: 10.1109/APSEC57359.2022.00039. [Online]. Available: https://ieeexplore.ieee.org/document/10023585

[13] Duc Minh Le et al., "Generating Multi-platform Single Page Applications: A Hierarchical Domain-Driven Design Approach," SoICT '22: Proceedings of the 11th International Symposium on Information and Communication Technology, 2022, doi: 10.1145/3568562.3568566. [Online]. Available: https://dl.acm.org/doi/10.1145/3568562.3568566

[14] Nicolò Pasini et al., "A virtual suturing task: proof of concept for awareness in autonomous camera motion," 2022 Sixth IEEE International Conference on Robotic Computing (IRC), 2023, doi: 10.1109/APSEC57359.2022.00039. [Online]. Available: https://ieeexplore.ieee.org/document/10023585

[15] P. Danenas and G. Garsva, "Domain Driven Development and Feature Driven Development for Development of Decision Support Systems," Information and Software Technologies, 2012, doi: 10.1007/978-3-642-33308-8_16. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-33308-8_16