ISSN: 2096-3246
Volume 58, Issue 03, 2026

REVERSE ENGINEERING–DRIVEN MAINFRAME MODERNIZATION: A STRUCTURED APPROACH TO API-LED TRANSFORMATION

Kalyana Sundaram Chidambaram
Independent Researcher, USA

Abstract
Mainframe modernization represents one of the most complex and consequential challenges in enterprise computing, driven by the need to integrate decades-old systems with contemporary digital architectures while preserving the validated business logic embedded within legacy codebases. Systems developed in languages such as COBOL and PL/1 encode institutional knowledge refined through years of operational use—knowledge that is rarely captured in formal documentation and would be extraordinarily difficult to reconstruct through redevelopment. Reverse engineering gives the ability to surface this logic, providing business logic, process flows, and data dependencies relevant for the organization in a systematic way. This involves using the source code and behavioral information instead of any incomplete system documentation artifacts available at hand. The uncovered business logic can then be wrapped up in independently deployable, REST-based service components with standardized interfaces and API-led architecture principles. JSON removes the format conversion overhead of most of the legacy middleware, and cloud hosting in turn allows for dynamic resource scaling, geographic redundancy, and built-in failover, making satisfying enterprise uptime and availability requirements much easier. DevOps and continuous integration further allow for the rapid and incremental promotion of code changes through the application without requiring time for a system-wide deployment to occur. The cumulative effect is an enterprise architecture vision that measurably improves performance, maintainability, scalability, and resilience without the disruption of the day-to-day operation and loss of institutional knowledge involved in wholesale replacement of systems. A preserve-first transformation framework offers the opportunity to stepwise modernize and extend legacy infrastructures into an agile, service-oriented architecture that is able to continuously innovate and adapt to changing business needs.
Keywords: Mainframe Modernization, Reverse Engineering, API-Led Architecture, Microservices, Legacy System Transformation

1. Introduction
For over fifty years, enterprise computing has been dominated by mainframe platforms providing transaction integrity, data consistency, and processing throughput for mission-critical workloads. Over time, these enterprise transactions have been improved with great depth of business logic representing the real-world regulatory context, domain knowledge, and institutional knowledge that is not typically documented in other business systems. The operational advantages of legacy systems are counterbalanced by their incompatibility with modern digital platforms, cloud infrastructure, and contemporary software development methods [1].

Other customary approaches to modernization have been to perform a wholesale swap-out of the codebase, either by reimplementing the system in a newer architecture in its entirety or retargeting the system as a whole onto a new platform. These carry risks including the need for thorough knowledge of business logic that is rarely documented, long projects with greater disruption, and loss of decades of refinement [2]. The net effect has been that many modernization projects were either delayed or over schedule or produced systems that only partially emulated the systems they were replacing.

Alternatively, a modernization approach based on reverse engineering does not discard the existing codebase. Rather, its goal is to enforce a structured analysis of the existing system, reconstruct and extract business rules, process flows, and the underlying data model, and store the existing institutional knowledge while permitting gradual architectural evolution [3]. When combined with API-led integration strategies leveraging modern cloud-based infrastructure, this provides a way to achieve a service-oriented architecture with a much lighter touch than a complete replacement.

We then describe four technical pillars we used to reverse engineer the legacy: reframing and reasoning with legacy applications as assets of business logic; dissecting the monoliths using dependency analysis; reconstructing reusable services by exposing APIs; and modernizing production deployments using meaningful cloud infrastructure and DevOps. In the final section, we summarize the strategy and the business implications of this convergence in terms of strengthening reliability, agility, and efficiency in the enterprise.

2. Legacy Systems as Assets of Business Logic

As usually encountered, reverse engineering-driven modernization is based on viewing some legacy codebases as not being a source of technical debt that is to be retired as quickly as possible (as is often the case in mainstream approaches) but systematically underestimating the business knowledge that they contain. The logic in mainframe systems, which have run successfully in production for decades, is not only achieved by the original design but also by the thousands of changes that have been made to deal with operational realities, regulatory changes, and corner cases as they have arisen over the years. The logic is essentially a validated behavioral specification, which would be extremely difficult to develop from scratch [1].

Analysis of reverse engineering reveals the inner workings and high-level logic of a system. Reverse engineering systematically disassembles source code, program execution traces, data flows, and interdependencies in order to reconstruct an accurate view of system operation [2]. In contrast, reverse engineering works with the artifact itself and helps to expose implicit dependencies, previously undocumented business rules, and behavioral assumptions that are encoded in the code but nowhere else [4].

It is most commonly applied to existing source code written in COBOL or PL/1 to identify indivisible units of functionality (e.g., routines that perform calculations, data transform routines, validate inputs, and check conditions for business rules) that can be transformed into reusable service-oriented components. Exposing such logic via RESTful APIs makes it available to multiple consuming applications within the enterprise that wish to use the same validated logic and avoid divergence. This is not merely a theory, as API exposure is a design pattern that is well established in the API design literature, concerned with the reuse of validated business logic to achieve integration without reimplementation.

Extraction Technique	Primary Input Source	Logic Types Recovered	Documentation Dependency
Static code analysis	Source code	Control flow, data structures	Low
Dynamic execution tracing	Runtime behavior	Process flows, edge cases	None
Dependency graph mapping	Program call trees	Inter-module relationships	Low
Data flow analysis	Variable propagation	Transformation rules, validations	Moderate

Table 1. Comparison of Legacy Business Logic Extraction Techniques [2, 4]

3. Dependency analysis for breaking monolithic architectures

By contrast, mainframe architectures tend to have more monolithic form factors, where the business logic, data access routines, and workflows are tightly coupled and may coexist in the same program units. Satisfying these coupling dependencies made sense in the early days of mainframe computers. It is a meaningful barrier to modularization. Untying these couplings is the technical challenge of the deconstruction phase [5].

Deconstruction usually begins with dependency analysis. This involves establishing functional dependencies between program units, shared data structures, sequences of execution, and control flow. Some components of the program may be decoupled and extracted as a functional unit, while some may be so interconnected with the rest of the code that they cannot be extracted without a major redesign. The output is a dependency map, the architectural blueprint, from which subsequent decomposition is derived [3].

The final concern at this stage is the removal of any intermediate communications layer that most legacy systems will have built up over the years. Many mainframe-based environments have a message queuing system and/or batch processing layer that is used for asynchronous processing and inter-system communication. However, while the mechanisms used to address the architectural challenges they encountered were valid at the time of their introduction, they add latency, increase the operational footprint due to additional points of failure, and incur operational overhead in the form of complexity related to both maintenance and incident handling [5]. Moving away from intermediate patterns towards the direct use of APIs, with services communicating over standardized, synchronous or asynchronous, HTTP-based interfaces [9], simplifies the architecture.

An advantage of using a reverse engineering-driven approach for stepwise modernization is that not all coupling needs to be resolved before starting the process of migrating to a new system. Functional units can often be isolated, extracted, and exposed as services, with the system continuing to operate while being modernized [2]. The same principle can be found in the literature on agile modernization. It often uses incremental delivery techniques as an alternative to big-bang replacement as a way to reduce risk [6].

4. Creating reusable API-exposed services from extracted business logic

Once identified using the dependency analysis, these functional units can be transformed into services in the transformation phase with deployable elements that provide standardized interfaces. All of the services will be independent and only provide specific business functionality through a respective RESTful API interface that can be used by any technology with an access right to it [8]. This encapsulation is by design: hiding implementation details allows the internal representation of the business logic, including whatever legacy constructs remain, to be iteratively refactored without breaking consumers [6].

The choice to use REST for the interface design pattern and JSON for the data format was guided by practical considerations as well as basic architectural principles: the resource-oriented nature of REST combined with the reuse of standard HTTP verbs within a common interface model greatly reduces integration complexity when working with heterogeneous systems [10]. JSON, with its human-readable, lightweight data format, is well suited to cross-platform data exchange, including integration with legacy backends for web or mobile clients [7]. This leads to an important data transformation overhead reduction when compared to previous patterns where data used to be serialized in proprietary formats multiple times through different layers .

From a service design perspective, a well-designed decomposition minimizes the coupling between different services and maximizes the cohesion within a service. Services should be decomposed along business capability (externally visible unit of value) boundaries, not along technical implementations. Services that align with a real business capability (such as eligibility, rate, or transaction validation) are better candidates for reuse and are more stable than those that align with an arbitrary technical partition [11]. Aligning service boundaries with business domain boundaries is core to the microservices pattern and is well supported in API design literature [12].

A standardized API contract serves a governance purpose, and businesses may use API management frameworks to retain control over global policies such as authentication, authorization, rate limiting, and service watching to address enterprise integration issues. This is especially true when the number of applications consuming the API increases [13]. Neither of these forms of governance is simply administrative , but rather an essential prerequisite to the type of common API reuse that makes the modernization investment worthwhile [14].

Design Dimension	Principle	Relevance to Legacy Transformation
Resource orientation	Map endpoints to business capabilities	Aligns service scope with extracted logic units
Data exchange format	JSON for lightweight interoperability	Eliminates legacy proprietary format dependencies
Interface contract stability	Versioned, backward-compatible APIs	Protects consumers during iterative refactoring
Endpoint granularity	Capability-scoped, not code-scoped	Prevents over-decomposition of tightly coupled routines

Table 2. RESTful API Design Considerations for Legacy-Extracted Services [8, 10, 12]

5. Cloud-based Operationalization and DevOps Enablement

API-led decomposition can only reach its full potential once the APIs and services are deployed in environments that can provide the scale, reliability, and operational control to meet the requirements of enterprise workloads. This environment is provided by cloud computing platforms. In contrast, dynamic resource allocation allows scaling the service in response to changes in service demand without manual work; geographical redundancy can improve high service availability; and managed infrastructure services reduce the overhead of managing the underlying compute and network infrastructure itself [7].

Microservices architecture conforms to the cloud computing model, as each microservice is scoped independently, which enables each microservice to be deployed, scaled, and updated independently of the others. This has operational implications since, unlike a monolithic application, any changes to one microservice do not require the redeployment of the entire application, and the failure of a single service will not affect the entire application [5]. Cloud providers enable this vision because they provide functionality to automate rerouting of requests to other healthy instances and to restore service by starting the workload in an alternate instance or an entirely different region with minimal operator involvement, which allows the core availability requirements of the mission-critical applications that mainframes were supporting to be met [9].

The DevOps team needed to deploy these services and also changed the operations model. Continuous integration and continuous deployment enable teams to confidently deploy changes to services on a rapid basis, as changes to live code take advantage of automated tests to validate the change before making it to production. In contrast to monolithic legacy systems, which require manual regression tests and coordinated release windows [3], automated testing, infrastructure-as-code, and continuously deployed pipelines enable iterative development and support the goal of moving quickly to respond to business needs without creating additional technical debt [6].

Likewise, agile practices further reinforce this approach at the team and process level. Agile practices often consist of short iterations through the development process with regularly delivered, visible working software, ensuring that modernization progress is made concrete at every engagement milestone rather than relegated to a distant go-live date. This increased visibility is particularly

important for scenarios such as mainframe modernization, where the larger and more complex the workload, the more difficult it is to determine whether it is moving correctly [2]. Fewer layers and increased architectural simplicity can also take operational maintenance load off the team, reducing the need to monitor, diagnose, and resolve production issues manually [15].

Capability	Description	Operational Benefit
Dynamic resource allocation	On-demand compute scaling	Handles variable transactional workloads without over-provisioning
Geographic redundancy	Multi-region service replication	Sustains availability during regional infrastructure failures
Automated failover	Traffic rerouting on instance failure	Minimizes downtime without manual intervention
Managed infrastructure services	Platform-handled patching and provisioning	Reduces operational overhead for modernization teams

Table 3. Cloud Deployment Capabilities Supporting Modernized Service Architectures [7, 9]

6. Planned Impact: Reliability, Scalability, and Enterprise Agility

The net effect of the combined concepts described in the previous sections is the transformation of system architecture and enterprise technical capability. Organizations that successfully adopt a reverse engineering-driven modernization strategy realize a number of planned benefits that go beyond technical improvements in individual systems.

The first result is increased reliability. Because there are no intermediary communication layers or intermediate points of failure and because the services are deployed on cloud infrastructure with built-in fail-over and redundancy, the architecture can be made more reliable and available than the systems it replaces because of these factors [9]. This should not be a surprise, as high availability is one of the main drivers for mainframe modernization, and microservices plus cloud directly address this challenge [5].

The second outcome is scalability. In monolithic mainframe systems, if a single component requires more resources, the entire mainframe may need to be upscaled as well to meet demand. Microservices can be independently scaled according to the real-time demand pattern, which is a more efficient utilization of resources and thus can have direct cost implications, especially for organizations that have large transactional workloads [7] .

Thirdly, enterprise agility, which focuses on providing business capabilities, external integration, and responsiveness to change in a way that avoids any disruption to business-as-usual operations. API-led architecture is key to achieving enterprise agility by exposing business functionality via standard interfaces. This forms an integration fabric whereby new applications can access existing services without modifying their code [14]. It is a critical component of digital transformation initiatives, which require rapid integration of new digital channels, third-party services, and emerging technologies [15].

Finally, because the approach is incremental, the above outcomes are achieved step by step rather than in one risky, big-bang event. Each extraction and exposure of a legacy function as an API is a step towards the final architecture. The legacy system continues to run while the new solution emerges. The balance of trade-off between stability and innovation is arguably the core value proposition of reverse engineering-driven modernization: the preservation of well-established business logic whilst incrementally disposing of the architectural constraints of legacy systems [1], [4].

Conclusion

Reverse engineering-driven modernization is a technically sound and operationally sustainable option for enterprises that wish to convert the monolithic process stacks of legacy mainframe environments into service-oriented architectures without the risks and added complexities imposed by wholesale replacement projects. By recognizing and retaining legacy-coded components as proven repositories of business logic, rather than technological 'dead weight' to be discarded, the design pattern provides a way to safely remove the architectural features (tight coupling between components, intermediary processing layers, and monolithic deployment patterns) that are preventing legacy systems from being effectively reused by modern digital infrastructures. By decoupling the discrete business capabilities from the legacy environments and packaging them as reusable RESTful, JSON-enabled service building blocks, the resulting integration fabric enables interoperability across heterogeneous technology environments and the rapid introduction of new digital capability without disrupting existing capability. When delivered in the cloud, the architecture enables the elastic capacity, geo-resilience, and automated failover considered essential for enterprise-grade availability and resilience, and enables the iterative service improvement at a cadence unattainable with legacy deployment models. Net effects like these can be achieved stepwise, not by heroic one-off transformations, but are available to organizations at different levels of modernization maturity. Organizations that adopt a preservation-first posture create not only a more powerful technical platform but also a more durable and flexible integration architecture than those who rely on disruptive re-platforming to accommodate the continuous shifts in technology, regulation, and business strategy that characterize this new world.

References

[1] 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

[2] Harry Sneed, "Planning the reengineering of legacy systems," IEEE Software, 1995. [Online]. Available: https://www.researchgate.net/publication/3247037

[3] Michigan Technological, "What is Software Engineering?" [Online]. Available: https://www.mtu.edu/cs/undergraduate/software/what/

[4] Jukka Viljamaa, "Reverse engineering framework reuse interfaces," ACM SIGSOFT Software Engineering Notes, 2003. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/949952.940101

[5] J. Bisbal et al., "Legacy information systems: issues and directions," IEEE Software, 1999. [Online]. Available: https://ieeexplore.ieee.org/document/795108

[6] Martin Fowler, “Patterns of Enterprise Application Architecture,” O'Reilly, 2002. [Online]. Available: https://www.oreilly.com/library/view/patterns-of-enterprise/0321127420/

[7] Divya Kaira, "Api-Led Connectivity: Enabling Agile And Scalable Digital Integration," International Research Journal of Modernization in Engineering Technology and Science, 2023. [Online]. Available: https://www.irjmets.com/uploadedfiles/paper//issue_6_june_2023/41925/final/fin_irjmets1686564511.pdf

[8] Li Li et al., "Design Patterns and Extensibility of REST API for Networking Applications," IEEE Transactions on Network and Service Management, 2016. [Online]. Available: https://www.researchgate.net/publication/290509847

[9] Renu Mishra et al., "Establishing a three-layer architecture to improve interoperability in Medicare using smart and strategic API-led integration," SoftwareX, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711023000729

[10] APITCHAKA SINGJAI et al., "Patterns on Designing API Endpoint Operations," in Proceedings of Conference on Pattern Languages of Programs (PLoP’21), 2021. [Online]. Available:

https://eprints.cs.univie.ac.at/7194/1/patterns-on-designing-api-endpoint-operations-proceeding-version5.pdf
[11] Jennifer Horkoff et al., "Strategic API Analysis and Planning: APIS Technical Report," arXiv preprint arXiv:1911.01235. [Online]. Available: https://arxiv.org/pdf/1911.01235
[12] Madhu Garimilla, "The Art Of Api Design: Best Practices For Modern Software Development," International Journal of Engineering and Technology Research (IJETR), 2024. [Online]. Available: https://iaeme.com/MasterAdmin/Journal_uploads/IJETR/VOLUME_9_ISSUE_2/IJETR_09_02_021.pdf
[13] Suffiyan Ul Hassan Farooqui, "API Management Challenges for EI," ResearchGate, 2024. [Online]. Available: https://www.researchgate.net/publication/382624105
[14] S. Chaudhari, "API-Led Connectivity: Architecting Modern Enterprise Integration," International Journal Of Information Technology And Management Information Systems, 2025. [Online]. Available: https://www.researchgate.net/publication/389326692
[15] Ankur Agarwal, "Pushing digital transformation through API-led connectivity," Nagarro, 2022. [Online]. Available: https://www.nagarro.com/en/blog/how-api-led-connectivity-accelerates-digital-transformation