The Eastasouth Journal of Information System and Computer Science
Vol. 2, No. 01, August 2024, pp. 101-119
ISSN: 3025-566X, DOI: 10.58812/esiscs.v2i01

Decoupling the Core: A Technical Roadmap for Modernizing Mainframe into Cloud-Native Microservices on Azure Kubernetes Service

Tathagata Roy
Independent Researcher, Colorado, USA

Article Info

Article history:
Received Aug, 2024
Revised Aug, 2024
Accepted Aug, 2024

Keywords:
API-Driven Development;
Automation Tools;
Azure Kubernetes Service (AKS);
CICS Modernization;
COBOL to Java;
Java Microservices;
Kafka, 3270 Web Conversion;
Mainframe Modernization;
Microsoft Entra ID;
Spring Batch;
Spring Boot;
Spring Data JPA

ABSTRACT

Modernizing tightly coupled mainframe ecosystems built on COBOL business logic, CICS transaction flows, CA7 batch schedules, and DB2/VSAM data structures remains a technically complex undertaking for enterprises seeking cloud-native agility. This paper presents a structured roadmap for decomposing monolithic mainframe workloads into Spring Boot microservices deployed on Azure Kubernetes Service, supported by Kafka for asynchronous, event-driven communication. The work examines the translation of COBOL Copybooks into JSON or Avro schemas for API-driven interoperability, the extraction of COBOL and stored-procedure logic into Spring Data and service layers, and the re-engineering of JCL utilities into containerized batch pipelines orchestrated through Kubernetes-native schedulers. It also evaluates approaches for transforming 3270 CICS interfaces into modern React or Spring Thymeleaf frontends, migrating IBM MQ patterns into Kafka-aligned messaging, and integrating RACF-based security models with cloud identity providers. Automated code-analysis and refactoring tools available at the time of writing are assessed for their ability to accelerate large-scale modernization while preserving transactional integrity and regulatory compliance. The resulting roadmap provides a technically grounded strategy for decoupling mainframe cores and transitioning toward resilient, modular, and cloud-aligned architectures.

Corresponding Author:
Name: Tathagata Roy
Institution: Independent Researcher, Colorado, USA
Email: tatha.roy@gmail.com

1. INTRODUCTION

Legacy mainframe platforms continue to support mission critical operations in high-volume, regulated industries such as financial services and healthcare. Their reliability, transactional integrity, and deterministic execution have made them central to enterprise computing for decades. These systems process large volumes of business transactions through COBOL programs, CICS regions [6], JCL-driven batch cycles, and tightly structured data models in DB2 [9] and VSAM [8]. While this architecture has delivered exceptional stability, its monolithic design and proprietary interfaces limit the adoption of modern integration patterns, distributed processing models, and rapid delivery practices [1].

The modernization imperative is driven by the growing mismatch between

mainframe development processes and the expectations of cloud-native engineering [21]. Decades of procedural logic, tightly coupled transaction flows, and waterfall-oriented release cycles create bottlenecks that impede the introduction of new capabilities [30]. Organizations also face increasing operational risk as the availability of mainframe specialists declines. In contrast, cloud platforms provide elastic scaling, automated deployment pipelines, distributed messaging [15], and API-driven interoperability. This disparity has accelerated the need to transition from monolithic mainframe ecosystems to modular, service-oriented architectures [29].

The objective of this paper is to define a structured roadmap for decomposing mainframe workloads into a Java 17 and Spring Boot based microservices architecture deployed on Azure Kubernetes Service [27]. The approach emphasizes pattern-based extraction of COBOL logic [5], transformation of CICS interactions into modern service interfaces [7], and synchronization of data across hybrid environments during phased migration [4]. It also outlines mechanisms for federating legacy security constructs such as RACF with cloud identity providers [10], [11] and evaluates the role of Kafka in replacing legacy messaging patterns [16]. Together, these contributions provide a technically grounded foundation for organizations seeking to decouple their mainframe cores and transition toward a scalable and cloud-aligned architecture.

2. BACKGROUND AND MOTIVATION

2.1. *Mainframe Technical Characteristics*

Mainframe architectures are engineered for high-volume transaction processing and deterministic throughput, with CICS serving as the primary transaction manager [6]. CICS executes application logic through a task-driven model in which terminal interactions are defined using Basic Mapping Support and rendered through 3270 interfaces [7]. Application logic is typically implemented in procedural COBOL or Assembler and employs pseudo-conversational techniques to manage state efficiently across large user populations. Offline workloads are defined through JCL and orchestrated by enterprise schedulers such as CA7, enabling multi-step processing pipelines that support critical end-of-day reconciliation and data transformation [1]. Data persistence relies on DB2 relational schemas [9] and VSAM file structures [8]. Although these provide high-speed indexed access, they are tightly coupled to application logic and lack the architectural flexibility required for distributed or document-oriented data paradigms.

2.2. *Drivers for Change*

Modernization efforts are driven by the need for elastic scalability, broader integration capabilities, and the adoption of modern event-driven processing [21]. Mainframes excel at vertical scaling but incur significant operational costs when handling the unpredictable traffic patterns common in digital channels. While legacy systems rely on IBM MQ for asynchronous communication and change data capture, modern cloud platforms provide superior horizontal scaling through container orchestration [27]. The transition from traditional message queuing to event-streaming platforms such as Kafka enables higher throughput and decoupled service interactions [15]. Kafka's distributed partitioning and log-based persistence [16] offer architectural advantages over traditional queuing models, supporting real-time analytics and complex event processing that are

difficult to scale within conventional mainframe constraints.

2.3. *The Agility Gap*

A significant challenge in mainframe-centric organizations is the disparity between legacy development cycles and cloud-native engineering practices [29]. Mainframe environments typically rely on sequential, waterfall-oriented release processes that require extensive coordination across COBOL programs and CICS regions. These cycles limit the ability to deliver incremental changes and respond to market demands with velocity. In contrast, microservices deployed on Azure Kubernetes Service benefit from container isolation and automated CI/CD pipelines [30]. The adoption of Kubernetes-native practices [20] enables rapid iteration and controlled rollout strategies such as blue-green deployments. This shift in operational velocity is essential for reducing technical debt and improving developer productivity in a landscape where speed to market is a primary competitive differentiator.

3. TECHNICAL TAXONOMY: LEGACY MAINFRAME PATTERNS

3.1. *Transaction Management*

CICS regions form the core execution environment for online mainframe applications, providing isolated address spaces that manage large volumes of concurrent transactions [6]. User interactions are defined through 3270 Basic Mapping Support maps, which specify screen layouts and field attributes for terminal sessions [7]. CICS employs a pseudo-conversational model that optimizes resource utilization by terminating a task after sending a screen to the user and storing state in a communication area or temporary storage queue. When the user responds, a new task is initiated and the state is restored. This model enables efficient scaling across thousands of users but introduces complexity when translating these interactions into stateless web protocols used in modern microservices architectures.

3.2. *Logic and Data Handling*

Business logic on the mainframe is primarily implemented in procedural COBOL, where control flow is organized into paragraphs and sections rather than modular components. Data structures are defined through Copybooks that specify fixed-width record layouts, often incorporating constructs such as REDEFINES clauses for memory reuse and OCCURS clauses for array definitions [5]. These structures tightly couple data representation with program logic. Performance critical routines are sometimes implemented in Assembler, embedding low-level operations optimized for the z/Architecture but difficult to interpret or refactor [1]. The combination of procedural logic, embedded data definitions, and low-level routines creates significant challenges when migrating to cloud-native systems that require clear separation between domain logic, data transfer objects, and persistence models [2].

3.3. *Batch and Orchestration*

Batch processing is orchestrated through JCL, which defines program execution steps, file allocations, and utility operations such as sorting and copying. These jobs are scheduled and coordinated by tools such as CA7, which manage extensive dependency chains based on completion codes, time windows, and event triggers [1]. Batch

workflows often support end-of-day reconciliation and reporting processes that must be completed within strict operational windows. Delays in any step can propagate through the dependency chain, affecting downstream processes. Modernizing these workflows requires decomposing monolithic, file-based sequences into distributed, event-driven pipelines that can execute in parallel and respond dynamically to workload conditions [30].

3.4. Persistence Layers

Mainframe data storage spans both relational and non-relational models. VSAM provides high-performance record storage through Key Sequenced Data Sets for indexed access and Entry Sequenced Data Sets for sequential logging [8]. These structures do not enforce referential integrity, placing responsibility for consistency on application logic. DB2 offers a relational environment with schema-driven integrity constraints and stored procedures that encapsulate complex SQL operations within the database engine [9]. Migrating these persistence models to cloud-native platforms requires mapping hierarchical or fixed-width VSAM records to relational or NoSQL schemas and adapting concurrency and locking mechanisms to distributed data stores.

3.5. Middleware and Messaging

IBM MQ provides the primary messaging backbone for asynchronous communication in mainframe environments. It supports reliable delivery through local and remote queues and integrates with transactional workloads using two-phase commit protocols to synchronize message persistence with database updates. MQ messages often contain binary or EBCDIC-encoded payloads defined by Copybooks, requiring specialized parsing and conversion when interacting with external systems. While MQ offers strong reliability guarantees, its centralized topology can limit horizontal scalability. Transitioning to distributed streaming platforms requires rethinking message formats, delivery semantics, and processing patterns to support partitioning and consumer groups [15], [16].

3.6. Security and Identity

Security on the mainframe is governed by External Security Managers such as RACF or ACF2, which provide centralized authentication and granular authorization for datasets, CICS transactions, and system commands [1]. These systems maintain detailed access lists and resource classes that define user permissions at a fine-grained level. Authentication typically relies on user identifiers and passwords, with limited support for modern multi-factor or federated identity mechanisms. Migrating to cloud-native identity providers requires translating mainframe entitlements into role-based access control structures while preserving auditability and compliance requirements [14].

4. MODERNIZATION STRATEGIES AND TRADE-OFFS

4.1. Rehosting (Lift and Shift)

Rehosting moves mainframe applications to distributed infrastructure or cloud-based emulators with minimal changes to COBOL or Assembler code [1]. This approach enables rapid reduction of mainframe operating costs and

allows organizations to exit proprietary hardware quickly [4]. Because the application logic remains intact, functional risk is low and operational continuity is preserved. The primary limitation is that the monolithic architecture remains unchanged, preventing the adoption of cloud-native capabilities such as horizontal scaling, modular deployments, or managed services. Rehosting improves the cost model but does not resolve agility constraints or the long-term dependency on legacy skills.

4.2. Refactoring (Phased Strangler Fig)

Refactoring through the Strangler Fig pattern focuses on gradually replacing mainframe components with cloud-native services [3]. New functionality is implemented as independent microservices, while existing logic is intercepted and redirected to these modernized components [29]. This reduces migration risk and enables incremental delivery of value. The coexistence period introduces challenges around maintaining data consistency, often requiring dual-write mechanisms or real-time change data capture pipelines [28]. Although more complex than rehosting, this strategy directly addresses technical debt and supports a controlled transition toward a domain-driven architecture [18].

4.3. Re-architecting (Cloud-Native Microservices)

Re-architecting involves decomposing the monolithic mainframe system into domain-aligned microservices implemented using modern frameworks such as Java and Spring Boot [2]. This requires rethinking procedural COBOL logic in terms of object-oriented and event-driven patterns, redesigning data models, and adopting distributed systems practices [19]. Although the initial investment is significant, this strategy provides the highest long-term flexibility. It enables independent deployment, automated resilience, and seamless integration with observability, security, and DevOps toolchains [30]. For organizations seeking sustained agility and rapid innovation, re-architecting offers the strongest architectural foundation.

4.4. Hybrid Modernization Models

Hybrid models combine elements of rehosting, refactoring, and re-architecting to balance cost, risk, and modernization speed [1]. Performance-critical or data-intensive workloads may remain on the mainframe or an emulator, while engagement layers, integration services, and analytics components are modernized into microservices. This approach is suitable when data gravity, regulatory constraints, or extreme transaction volumes make full migration impractical [4]. By integrating the mainframe with cloud-native services through event streaming and high-speed connectors, organizations can preserve mainframe reliability while gaining cloud-native agility [15].

4.5. Decision Framework

Selecting the appropriate modernization strategy requires evaluating system complexity, organizational risk tolerance, regulatory requirements, and data gravity [4]. Highly coupled systems with extensive batch dependencies may benefit from phased refactoring, while well-bounded domains can be re-architected more directly [18]. The cost and latency of moving large datasets influence whether workloads should be migrated, modernized in place, or

integrated through hybrid patterns. The availability of modernization tools and the skill set of the engineering team also shape the approach [5]. A structured assessment of these factors enables architects to prioritize workloads based on business value and technical readiness.

5. PATTERN CONVERSION: FROM Z-ARCHITECTURE TO MICROSERVICES

5.1. CICS to REST and Backend-for-Frontend

Modernizing CICS-based 3270 interactions requires decomposing screen-driven flows into stateless REST APIs [6]. In the mainframe environment, CICS manages pseudo-conversational state through Basic Mapping Support and communication areas, which must be reinterpreted as discrete service calls [7]. A Backend-for-Frontend pattern is typically introduced to aggregate multiple domain services and tailor responses for React applications or Thymeleaf templates [29]. This approach encapsulates legacy navigation logic within an orchestration layer, allowing the frontend to operate independently of mainframe constraints while supporting scalable, stateless communication. By shielding the presentation layer from COMMAREA and Temporary Storage Queue complexities, the BFF ensures a modern user experience without direct dependency on legacy session management.

5.2. COBOL Logic to Java 17

Migrating COBOL logic to Java 17 requires restructuring procedural flows into domain-driven components rather than performing direct syntax translation [18]. COBOL paragraphs and control structures are reorganized into service classes, domain entities, and application layers within Spring Boot [2]. Java 17 features such as records for immutable data carriers and sealed classes for domain modeling help create a type-safe, object-oriented representation of legacy business rules. This refactoring isolates business logic from infrastructure concerns, improves testability through JUnit and Mockito, and aligns the application with modern modular design principles [19]. The transition also enables the use of functional streams to replace procedural loops, enhancing code maintainability and readability.

5.3. Copybook to Schema Conversion

Copybooks define the physical, fixed-width data structures that must be converted into JSON or Avro schemas for cloud-native interoperability [5]. This conversion must account for mainframe-specific types such as packed decimals (COMP-3), EBCDIC character encoding, and complex REDEFINES clauses that allow multiple interpretations of a single memory block. Automated transformation tools are often used to generate initial Java POJOs and schemas, though manual refinement is typically required to ensure semantic accuracy and preserve mainframe arithmetic precision. The resulting schemas provide a consistent contract for service communication, enabling seamless integration between modernized microservices and the remaining mainframe environment during the migration lifecycle [15].

5.4. Data Access Modernization

Modernizing the data layer involves migrating DB2 queries and VSAM file operations to Spring Data JPA repositories [9]. Cursor-based

processing commonly found in COBOL programs is replaced by pagination, streaming, or batch retrieval patterns that align with relational or NoSQL data stores. VSAM datasets require careful mapping to relational models or document-oriented structures, where indexed VSAM keys (KSDS) are translated into primary keys or unique identifiers in the target store [8]. Spring Data's abstraction helps separate domain logic from persistence mechanics, enabling horizontal scaling and improved portability across cloud database platforms such as Azure SQL or Cosmos DB while maintaining referential integrity previously enforced by application logic.

5.5. Batch Modernization

JCL-driven batch workflows are reimplemented using Spring Batch, which provides structured support for tasklets, chunk-oriented processing, and checkpointing [30]. Multi-step JCL jobs are mapped to Spring Batch job definitions, with ItemReaders, ItemProcessors, and ItemWriters replacing legacy utilities for sorting and transformation. Scheduling is transitioned from CA7 to Kubernetes-native orchestration using AKS CronJobs, enabling container-level resource isolation and parallel processing [27]. This approach preserves the deterministic processing semantics required for large-volume reconciliations while supporting integration with event-driven triggers and cloud-native observability tools [20].

5.6. Stored Procedure Extraction

DB2 stored procedures often contain embedded business logic that must be externalized to achieve true microservice independence [9]. Extracting this logic into Spring Boot service layers improves version control, observability, and scalability by moving compute-intensive rules from the database to the application tier. In cases where data-intensive operations must remain close to the database during transition, native queries or repository-based procedure calls serve as intermediate mechanisms. Over time, these procedures are decomposed into domain services that align with a bounded-context architecture, reducing database lock contention and facilitating the move toward polyglot persistence [18].

5.7. Error Handling and Transaction Boundaries

Distributed systems require transaction management strategies that differ from the centralized two-phase commit protocols used on the mainframe [2]. Modernized microservices implement idempotent operations, retry logic, and the Saga pattern to maintain consistency across service boundaries [17]. Saga-style workflows replace mainframe syncpoints by coordinating state changes through event-driven compensation logic [29]. Spring's declarative transaction management facilitates the definition of clear boundaries within each service, providing reliability comparable to CICS transactions while supporting the high availability and scalability of a cloud-native environment.

Table 1. Mainframe components to microservice patterns

Mainframe Component	Java/Spring Equivalent	Key Transformation Pattern
COBOL Programs	Spring Boot Services, Java 17 Record Classes	Procedural → Object-Oriented, Domain-Driven Design
CICS Transaction Server	Spring MVC REST	Pseudo-conversational →

Mainframe Component	Java/Spring Equivalent	Key Transformation Pattern
	Controllers, Backend-for-Frontend (BFF)	Stateless HTTP/HTTPS
3270 BMS Maps	React/Spring Thymeleaf, HTML5/JavaScript	Terminal UI → Web UI, Screen-flow → SPA
COBOL Copybooks	JSON/Avro Schemas, Java POJOs/Records	Fixed-width → Self-describing, EBCDIC → UTF-8
DB2 for z/OS	Spring Data JPA, Azure SQL/PostgreSQL	Direct SQL → Repository, Stored Procedures → Services
VSAM (KSDS/ESDS)	Spring Data JPA, Azure Cosmos DB	Indexed files → Relational, Hierarchical → Document
JCL Batch Jobs	Spring Batch, AKS CronJobs	Step-based → Chunk-oriented, File-based → Event-driven
CA7 Scheduler	Kubernetes CronJobs, Azure Logic Apps	Centralized → Distributed, Dependency chains → Events
IBM MQ	Apache Kafka, Spring Kafka	Queue-based → Log-based, Point-to-point → Pub-sub
RACF/ACF2	Microsoft Entra ID, PingOne/Okta	Resource profiles → RBAC, Fixed IDs → OIDC/OAuth2
CICS Syncpoint	Spring @Transactional, Saga Pattern	Two-phase commit → Saga, Centralized → Distributed
CICS COMMAREA	REST Request/Response, JWT Claims	Binary → JSON, Session state → Stateless
DB2 Stored Procedures	Spring Service Layer, JPA Entity Managers	Database logic → App logic, Centralized → Distributed
CICS Temporary Storage	Redis/Azure Cache, Kafka Streams State Stores	Memory queues → Distributed, Single-node → Multi-node
SMF/RMF Monitoring	Prometheus/Grafana, Azure Monitor	Batch logs → Real-time, Centralized → Distributed

6. TARGET ARCHITECTURE: AZURE KUBERNETES SERVICE

6.1. AKS Cluster Architecture and Infrastructure

Azure Kubernetes Service provides the managed orchestration layer for deploying the modernized microservices [27]. The cluster is organized into separate node pools to isolate system components from application workloads. System node pools run core services such as DNS and metrics, while user node pools host business domain services and scale through Virtual Machine Scale Sets. Networking is implemented with Azure CNI and Cilium to support high-performance pod communication and integration with virtual networks. This model enables granular network policies and efficient IP management, which is essential when transitioning from the centralized address space of the mainframe to a distributed environment [20].

6.2. Deployment Topology and Service Discovery

The deployment topology follows a microservices pattern in which each bounded context extracted from the mainframe is deployed as a containerized service with multiple replicas [29]. Kubernetes DNS provides internal service discovery, allowing services to communicate through stable names rather than dynamic IP addresses [20]. External traffic is routed through an API gateway such as Azure Application Gateway with AGIC or an NGINX Ingress Controller. The gateway centralizes SSL termination, routing, and

security controls, creating a boundary similar to the CICS front-end processor while supporting modern web and mobile access patterns [21].

6.3. *Service Mesh Considerations*

A managed Istio service mesh is used to standardize service-to-service communication [27]. Sidecar proxies intercept traffic to provide consistent handling of retries, timeouts, and routing rules. This enables controlled rollout strategies such as canary and blue-green deployments, which are important when validating modernized logic alongside legacy components [30]. Mutual TLS is enforced across all service communication, replacing the implicit trust of mainframe memory-based interactions with a zero-trust model suited to distributed systems [14].

6.4. *Observability and Monitoring*

Distributed systems require a comprehensive observability stack to replace the centralized diagnostics available on the mainframe [22]. OpenTelemetry provides tracing across microservices and integrates with Azure Monitor and Application Insights to identify latency issues and failures. Logs are aggregated into a Log Analytics Workspace for correlation and analysis using Kusto Query Language. Metrics from the cluster and applications are collected through a managed Prometheus service and visualized in Azure Managed Grafana [23], [24]. This unified approach restores the operational visibility needed for high-volume transactional workloads.

6.5. *Configuration and Secrets Management*

Configuration and secrets are externalized to improve security and portability [30]. ConfigMaps store non-sensitive settings, while sensitive values are managed through Azure Key Vault integrated with the Secrets Store CSI Driver [26]. This allows secrets to be mounted securely without embedding them in images or manifests. Azure Workload Identity provides short-lived tokens for pod authentication to cloud services, and automated secret rotation ensures credentials remain current without manual intervention [25]. These practices strengthen security while simplifying operational management.

6.6. *High Availability and Resilience*

High availability is achieved by distributing node pools across multiple availability zones, ensuring that a zone-level failure does not disrupt service continuity [27]. Pod topology spread constraints ensure that replicas are evenly distributed across nodes and zones [20]. Pod Disruption Budgets define minimum availability during upgrades or maintenance. Autoscaling mechanisms, including the cluster autoscaler and horizontal pod autoscaler, adjust capacity based on workload demand. Together, these features create a resilient execution environment capable of supporting the high-volume, low-latency requirements of modernized mainframe workloads.

7. EVENT-DRIVEN MIGRATION AND MIDDLEWARE MODERNIZATION

7.1. *MQ-to-Kafka Migration and Message Transformation*

The transition from IBM MQ to Apache Kafka represents a fundamental shift from traditional point-to-point queuing to a distributed event-streaming paradigm [15]. In the legacy

environment, IBM MQ provides transactional message delivery with destructive reads, which is replaced by Kafka's immutable append-only log [16]. This migration requires a structured topic-mapping strategy in which MQ queues are translated into Kafka topics, with careful consideration given to partitioning. To maintain strict message-ordering guarantees, events are routed to specific partitions using consistent hashing of business keys such as account or transaction identifiers. Message formats previously defined by COBOL Copybooks or custom binary layouts are transformed into Avro or JSON schemas [5]. This transformation ensures that legacy payloads are converted into modern, self-describing formats that support interoperability and long-term schema evolution while preserving original field semantics.

7.2. *Asynchronous Patterns and Distributed Consistency*

Mainframe workloads often rely on two-phase commit protocols to coordinate atomic updates across MQ and DB2. In a distributed environment, these blocking semantics are replaced by patterns that support eventual consistency [29]. The Transactional Outbox pattern ensures atomicity by writing both the business update and the outbound event in the same local transaction, with a relay process publishing the event to Kafka [2]. Multi-service workflows are coordinated using the Saga pattern, where each step emits an event that triggers the next action. Compensating transactions reverse partial updates when failures occur, providing reliability comparable to mainframe rollback behavior without centralized coordination [17].

7.3. *Real-time Synchronization via CDC*

Hybrid coexistence during a phased migration requires real-time data parity between the mainframe and the new services in AKS [4]. This is achieved through log-based Change Data Capture (CDC) pipelines that monitor DB2 transaction logs or VSAM updates and stream changes directly into Kafka topics without impacting source-system performance [28]. By utilizing CDC, organizations can maintain a synchronized read-model in the cloud, allowing modernized services to operate on near-real-time data. An alternative approach is to implement a consumer service in AKS that reads CDC events from MQ and republishes them to Kafka.

7.4. *Schema Evolution and Governance*

Interoperability between legacy EBCDIC-encoded structures and modern JSON or Avro formats requires strong schema governance [15]. Avro schemas are managed through a schema registry that enforces compatibility rules, allowing producers and consumers to evolve independently. Backward and forward compatibility ensure that new fields or refined data types do not break existing consumers. This versioning strategy provides a stable contract between services and supports the transition from fixed-width Copybook definitions to flexible cloud-native payloads.

7.5. *Throughput, Backpressure, and Consumer Lag*

Kafka consumers must be tuned to handle high-volume transactional workloads [16]. Batch-size and fetch-interval settings are adjusted to balance latency and throughput, reducing network overhead during peak loads. Backpressure mechanisms ensure that downstream services are not overwhelmed when event volume

spikes. Consumer lag is monitored through Prometheus to detect bottlenecks early and maintain processing continuity [23]. Kafka's log-based design also enables replay of historical events for recovery or state reconstruction, providing resilience capabilities not available in traditional destructive-read queuing systems [17].

8. SECURITY INTEGRATION AND IDENTITY

8.1. RACF to Industry-Standard IdP Mapping

Migrating from RACF to modern identity providers such as PingOne or Okta requires translating legacy permissions into standardized RBAC structures [1]. RACF profiles that control access to datasets, CICS transactions, and system resources are mapped to IdP security groups and application roles [14]. User identities transition from fixed-length RACF IDs to globally unique identifiers, with entitlements consolidated into domain-aligned roles that support least-privilege access [18]. Synchronization connectors are typically used to extract RACF metadata and populate the IdP directory, ensuring continuity of access as users move from terminal-based workflows to distributed web and API environments.

8.2. Authentication and Authorization with OIDC and OAuth2

Authentication within the modernized microservices architecture is standardized on OpenID Connect (OIDC) [10] and OAuth2 protocols [11], replacing the proprietary sign-on mechanisms of legacy subsystems. The microservices act as OAuth2 Resource Servers, validating JSON Web Tokens (JWTs) [12] issued by the centralized IdP. For end-user interactions, the Authorization Code Flow with PKCE (Proof Key for Code Exchange) [13] is implemented to secure public and confidential clients. Authorization is enforced at the service level by evaluating JWT claims, where IdP-assigned roles are mapped to application-specific permissions. This ensures that every request is explicitly verified against a verifiable identity, replicating the deterministic security of the mainframe within a standards-based, interoperable framework.

8.3. Token Propagation and Service Identity

A Token Relay pattern preserves user context across microservice boundaries by forwarding the original bearer token in downstream calls [29]. This allows each service to apply fine-grained authorization based on the initiating user's claims. For background or system-initiated operations, the Client Credentials flow is used [11], enabling services to authenticate with their own machine identities. This dual model supports both user-driven and service-driven interactions while maintaining consistent authorization controls.

8.4. Workload Identity Federation

Workload Identity Federation secures access to cloud resources without relying on static secrets [27]. Microservices obtain short-lived OIDC tokens that are federated with the chosen IdP, allowing workloads to exchange these tokens for resource-specific access tokens at runtime. This eliminates long-lived credentials, reduces operational overhead, and aligns with zero-trust principles by enforcing identity-based access for each service instance [14].

8.5. API Gateway Enforcement and Zero-Trust Principles

The API gateway acts as the primary enforcement point for inbound traffic, performing token validation, rate limiting, and request filtering before forwarding requests to backend services [21]. Within the cluster, a service mesh enforces mutual TLS for all inter-service communication, ensuring that each connection is authenticated and authorized [27]. This layered approach replaces perimeter-based security with a zero-trust model that validates identity and intent at every hop [14].

8.6. Centralized Secrets and Key Management

Centralized secret management is required to replace the mainframe's built-in security model in a distributed environment [25]. Sensitive values such as credentials, certificates, and encryption keys are stored in a dedicated vault service, typically Azure Key Vault [26] or HashiCorp Vault, while Kubernetes Secrets handle non-critical or short-lived data [20]. The Secrets Store CSI Driver mounts the vault-managed secrets directly into pods at runtime, ensuring they are never embedded in container images or written to disk. Automated rotation policies refresh credentials without redeploying workloads, and centralized audit logs provide full traceability for compliance and operational oversight. This approach maintains strong cryptographic hygiene and consistent security controls across all microservices.

9. DATA MIGRATION AND COEXISTENCE

9.1. VSAM Migration and Record Transformation

Migrating VSAM datasets requires restructuring fixed-width, record-oriented files into relational or document-based models [8]. VSAM KSDS records, often containing nested OCCURS and REDEFINES clauses, are flattened or normalized to align with modern storage systems [5]. Primary keys are mapped to indexed columns to preserve lookup performance, while variable-length segments and hierarchical structures are represented through child tables or JSON documents. During this process, implicit business rules formerly embedded in the physical file layout are externalized into application logic, enabling the modernized system to leverage dynamic query capabilities and horizontal scaling that were previously restricted by the rigid mainframe filesystem [29].

9.2. DB2 Migration and Logic Extraction

DB2 migration focuses on converting schemas and extracting business logic embedded in stored procedures [9]. While relational structures map cleanly to cloud databases, mainframe-specific elements such as EBCDIC collation and complex views require adjustment [1]. Stored procedures that encapsulate business rules are reimplemented within microservices to improve independence, observability, and scalability [2]. Transitional support for native queries or repository-based procedure calls may be retained, allowing cloud services to connect directly to DB2. However, long-term modernization shifts compute-intensive logic from the database engine to the application

layer using modern Spring Data JPA abstractions.

9.3. *Dual-Write and Eventual Consistency Patterns*

Hybrid coexistence requires synchronization between the mainframe and modernized systems [4]. Dual-write patterns coordinate updates to both environments, while the Transactional Outbox pattern ensures reliable event publication to Kafka, as described in earlier sections [29]. Eventual consistency is maintained through deterministic conflict-resolution strategies such as timestamp ordering or optimistic locking [17]. These patterns allow both systems to operate concurrently without centralized transaction managers, supporting phased migration with controlled risk.

9.4. *Data Reconciliation and Parallel Execution*

Reconciliation ensures that modernized services produce results consistent with the mainframe during coexistence [30]. Record-level validation compares datasets across environments to detect discrepancies introduced by transformation logic or timing differences. Parallel-run strategies execute identical workloads on both systems, comparing outputs from COBOL programs and Spring Boot services to verify functional equivalence [5]. These validation pipelines provide the evidence required to confirm readiness for full cutover.

9.5. *Cutover Sequencing and Rollback Planning*

Cutover is executed through a controlled sequence designed to minimize operational risk [30]. A freeze window prevents data divergence during final synchronization, after which remaining deltas are applied through high-speed CDC pipelines [28]. Traffic is shifted gradually, beginning with read-only operations before enabling full write capability [4]. A rollback plan is maintained throughout the process to restore mainframe processing if issues arise. This structured approach ensures continuity of mission-critical operations during the transition.

10. AUTOMATION TOOLING ASSESSMENT

10.1. *AI-Assisted Refactoring and Transformation*

AI-assisted refactoring tools accelerate the initial stages of mainframe modernization by analyzing COBOL programs and generating functionally equivalent Java code [5]. IBM watsonx Code Assistant for Z supports application discovery by explaining complex modules, identifying cohesive logic, and producing preliminary Java implementations [1]. CloudFrame Renovate provides deterministic COBOL-to-Java transformation with strict preservation of numeric precision and execution semantics, generating Spring Boot–compatible output suitable for incremental modernization [5]. These tools reduce manual effort during early translation phases, allowing engineering teams to focus on architectural restructuring rather than line-level conversion.

10.2. *UI and Data Migration Tooling*

Modernizing the presentation layer without disrupting core logic is facilitated by 3270 connectors and screen-transformation utilities [7]. Nocode tools such as Virtel Screen Redesigner or Synchrony Systems Facelift enable non-invasive conversion of green-screen interfaces into modern web UIs by mapping Basic Mapping Support (BMS) maps to HTML5 and

JavaScript components [6]. These utilities allow rapid replacement of terminal-driven navigation with modern web features such as dropdowns and date pickers without modifying underlying mainframe code. On the data tier, migration is supported by specialized utilities that automate the conversion of VSAM datasets and DB2 schemas [8], [9]. These tools handle the complex translation of EBCDIC encoding and packed decimal formats (COMP-3) into cloud-compatible types, ensuring that historical data remains consistent and accessible within modern relational or document-oriented stores [4].

10.3. Messaging and Integration Adapters

Interoperability between the mainframe and the cloud-native ecosystem is maintained through robust messaging adapters that facilitate hybrid coexistence [4]. The IBM MQ Source and Sink connectors for Apache Kafka serve as a critical bridge, enabling bi-directional, "exactly-once" delivery of messages between legacy MQ queues and Kafka topics [15], [17]. These adapters support the transition by replicating queue messages into event streams, preserving ordering and metadata required for downstream processing. By providing out-of-the-box converters for binary and COBOL Copybook payloads, these integration tools simplify the task of mapping legacy data into modern JSON or Avro schemas [5].

10.4. Tooling Limitations and Manual Requirements

Despite their capabilities, automated tools have limitations that require architectural oversight [29]. Generated Java often resembles procedural JOBOL-style output, mirroring COBOL control flow rather than applying domain-driven design principles [18]. Manual refactoring is required to establish clear service boundaries, isolate business rules, and improve maintainability [19]. Automated tools also struggle with undocumented dependencies, Assembler routines, and implicit business logic, creating testing gaps that must be addressed through custom integration and regression suites [30]. High-fidelity conversion provides a correct baseline, but long-term modernization depends on deliberate restructuring and comprehensive CI/CD practices [30].

11. TESTING STRATEGIES FOR MAINFRAME MODERNIZATION

11.1. Functional Parity and Behavioral Equivalence

Ensuring functional parity between the modernized system and the legacy mainframe requires rigorous validation of behavioral equivalence [5]. Golden Master testing captures canonical outputs from COBOL programs and compares them against results produced by the modernized Spring Boot services. Bit-for-bit comparison is used for deterministic workloads such as financial calculations or regulatory reporting. This establishes a baseline for verifying that the new implementation preserves original semantics while supporting safe, incremental refactoring [30].

11.2. Copybook and Schema Validation

Copybook and schema validation ensures that data structures remain consistent across distributed environments [5]. Field-level validation compares each attribute in the transformed JSON or Avro schema with its corresponding COBOL definition, accounting for

packed decimals, sign representation, and EBCDIC encoding. Automated validators detect discrepancies introduced during schema conversion or serialization, ensuring that downstream microservices interpret data correctly and remain compatible with legacy consumers during hybrid operation [15].

11.3. *Batch Modernization and Restartability Testing*

Batch modernization testing verifies step equivalence and operational behavior within the Spring Batch framework [30]. Each JCL step is mapped to a corresponding tasklet or chunk-oriented component, with test harnesses validating that the modernized job produces identical outputs under equivalent inputs. Restartability and checkpoint logic are tested by simulating failures at controlled points to ensure that processing resumes without data loss or duplication, preserving the determinism expected of mainframe batch operations.

11.4. *Data Migration and Reconciliation*

Data migration and reconciliation testing validate the accuracy of VSAM and DB2 conversions through record-level comparison [8], [9]. Parallel-run validation executes business processes on both systems and compares resulting database states to confirm functional equivalence [5]. These tests identify discrepancies caused by encoding differences, field truncation, or transformation logic, reducing migration risk and providing evidence that the modernized system can assume full production responsibility.

11.5. *Event-Driven and Messaging Validation*

Event-driven validation ensures correctness in asynchronous workflows by testing ordering, replay behavior, and idempotency [17]. Kafka partition ordering is verified to maintain deterministic processing, while replay scenarios confirm state consistency during consumer restarts [15]. Throughput and consumer-lag testing evaluate the system's ability to sustain event volumes previously handled by MQ-based systems, ensuring reliability during hybrid coexistence [16].

11.6. *CICS UI and Hybrid Coexistence Testing*

CICS UI modernization testing validates screen-flow equivalence and preserves the behavior of 3270 interactions [7]. This includes verifying navigation sequences, PF-key behavior, cursor positioning, and field-level validation rules. Hybrid coexistence testing confirms dual-write behavior and ensures that CDC pipelines maintain synchronization between environments [28]. Eventual-consistency checks verify that data propagated through event streams converges to a consistent state across both systems during incremental migration [17].

11.7. *Cutover and Production Readiness Testing*

Cutover and production-readiness testing validate the system's ability to transition to full cloud-native execution [4]. Dry-run cutovers simulate the final migration sequence, including freeze windows, delta synchronization, and traffic redirection [30]. Observability validation ensures that distributed traces, logs, and metrics provide sufficient visibility for incident

response [22]. These tests confirm that the modernized system is ready for production traffic and that rollback procedures are well-defined and technically verified.

12. IMPLEMENTATION ROADMAP: THE PATH TO PRODUCTION

12.1. Foundation Phase

The modernization journey begins with the establishment of a robust cloud-native foundation designed to support mission-critical financial workloads [1]. This phase focuses on deploying an Azure Kubernetes Service landing zone, incorporating enterprise-grade networking via Azure CNI and Cilium for high-performance pod communication [27]. A critical objective is the implementation of identity integration, where legacy security constructs from RACF are federated with an industry-standard Identity Provider using OpenID Connect [10] and OAuth2 [11]. Security hardening is further achieved through the integration of Azure Key Vault for secrets management [26] and the establishment of a managed service mesh for mutual TLS and traffic observability [14]. This architectural baseline ensures that the target environment meets the stringent compliance and security requirements of the mainframe ecosystem before any application logic is migrated.

12.2. Pilot Phase

The pilot phase focuses on migrating low-complexity workloads to provide early validation of the platform, deployment pipelines, and transformation tooling [4]. Candidate applications typically include stateless services, reference-data lookups, or auxiliary batch processes with minimal dependencies on core mainframe subsystems. This phase serves as a proof of technology for the testing framework, specifically validating Golden Master verification and record-level reconciliation processes [5]. Successful execution builds stakeholder confidence and informs critical refinements to the migration framework before scaling the effort [30].

12.3. Execution Phase

The execution phase applies domain-driven design principles to incrementally migrate business capabilities using the Strangler Fig pattern [3], [18]. Individual domains are decomposed into bounded contexts, and new microservices implemented in Java 17 and Spring Boot are introduced alongside existing mainframe components [19]. Business logic is systematically extracted from the monolith, with transformation tooling handling high-volume code conversion while manual refactoring ensures architectural alignment [5]. Traffic is gradually redirected to the modernized services as they reach functional parity, enabling iterative delivery and continuous validation of business behavior [29].

12.4. Coexistence Phase

During the coexistence phase, the architecture supports hybrid operation in which the legacy mainframe and the cloud-native environment function as a unified system [4]. Change Data Capture (CDC) pipelines [28] and the Transactional Outbox pattern [2] synchronize updates between DB2/VSAM datasets and cloud databases. These mechanisms ensure data parity and allow modernized services to participate in real-time workflows while legacy components remain active.

Continuous reconciliation and parallel-run validation are performed during this phase to maintain data integrity and system stability across the hybrid ecosystem.

12.5. *Decommissioning Phase*

The decommissioning phase marks the transition from hybrid operation to full cloud-native execution and the retirement of legacy infrastructure [1]. Final cutover activities include established freeze windows, final delta synchronization, and the permanent redirection of all traffic to the modernized services. Once all domains have been validated and operational-readiness criteria are met, the mainframe workloads are retired [30]. This phase concludes with the decommissioning of the physical Z-hardware, archival of historical datasets for compliance, and the final stabilization of long-term operational processes for the cloud-native system.

13. CHALLENGES AND MITIGATION

13.1. *Data Consistency and Referential Integrity*

Maintaining data consistency during the transition from a monolithic mainframe to a distributed microservices architecture is a central challenge [2]. Mainframe environments rely on tightly coupled transaction managers, whereas cloud-native systems distribute state across independent services [29]. Consistency is maintained through Change Data Capture pipelines [28], the Transactional Outbox pattern, and idempotent consumers that ensure reliable event propagation during partial failures [17]. Automated reconciliation processes compare record-level states across environments to detect and correct divergence, preserving referential integrity throughout hybrid operation.

13.2. *Latency and Distributed System Overheads*

Distributed architectures introduce network latency and serialization overhead that differ from the deterministic performance of mainframe cross-memory communication [1]. Mitigation strategies include co-locating services within bounded contexts, applying caching layers to reduce repeated lookups, and optimizing payload formats to minimize serialization cost [18]. Service-mesh configurations are tuned to reduce proxy overhead while maintaining observability and security [27]. These techniques ensure that the modernized system meets performance expectations without compromising modularity.

13.3. *Event Ordering and Duplication*

Event-driven systems must address ordering and duplication challenges that do not exist in sequential mainframe processing [16]. Kafka provides ordering guarantees only within partitions, requiring careful selection of partition keys to ensure deterministic processing for entity-specific workflows [15]. Duplicate events may occur during retries or consumer restarts, making idempotent handlers essential [17]. These patterns ensure that distributed services maintain the correctness and predictability expected of transactional workloads.

13.4. *Organizational and Governance Challenges*

Modernization efforts span multiple domains and require alignment across architecture, security, and operations teams [1].

Legacy governance models often assume centralized control, whereas microservices rely on decentralized ownership and automated policy enforcement [29]. Establishing clear domain boundaries, standardized migration patterns, and platform-level guardrails ensures consistent implementation across teams [19]. Governance frameworks and automated compliance checks help maintain architectural integrity as the system evolves [30].

14. CONCLUSION

The modernization of mainframe systems into an event-driven [15], cloud-native architecture [21] requires a structured approach that balances technical rigor with operational continuity. Successful transformation depends on automated refactoring [5], domain-driven decomposition [18], hybrid coexistence patterns [4], and comprehensive testing strategies [30]. These elements ensure that functional parity, data integrity, and system reliability are preserved throughout the migration lifecycle. The integration of AI-assisted refactoring and high-fidelity conversion engines accelerates this lifecycle by automating the discovery of legacy logic and producing deterministic translations into modern frameworks [1], allowing engineers to focus on architectural alignment and domain boundaries [19].

Beyond the transition from COBOL to microservices, the adoption of distributed data models and container-oriented deployment [20] introduces improved patterns for scalability, resilience, and observability [22]. These capabilities enable integration with modern ecosystems and support continuous delivery practices that were previously impractical in monolithic environments [30]. By embracing event streams [16], idempotent processing [17], and decentralized state management [2], organizations can build systems that respond effectively to real-time business demands while reducing long-term operational complexity [29].

The processes described in this paper represent a generalized framework rather than a prescriptive sequence. Each organization must adapt these practices to its own constraints, legacy landscape [1], and regulatory requirements. Variations in data models and workload characteristics will influence the specific migration path, but the principles remain broadly applicable for designing a modernization strategy that balances risk, cost, and long-term architectural value.

REFERENCES

[1] IBM Corporation, Armonk, NY, USA, "Mainframe Modernization on the IBM z16: Strategy and Infrastructure," IBM Redbooks, Document SG24-8530-00, 2023. [Online]. Available: https://www.redbooks.ibm.com/abstracts/sg248530.html [Accessed: July 2024]

[2] C. Richardson, Microservices Patterns: With Examples in Java. Shelter Island, NY, USA: Manning Publications, 2018.

[3] M. Fowler, "StranglerFigApplication," martinfowler.com, 2004, updated 2023. [Online]. Available: https://martinfowler.com/bliki/StranglerFigApplication.html [Accessed: July 2024]

[4] Microsoft Corporation, Redmond, WA, USA, "Mainframe Migration Overview," Azure Architecture Center, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/mainframe/guide/ [Accessed: July 2024]

[5] CloudFrame, Inc., San Francisco, CA, USA, "Automated COBOL to Java Transformation for Cloud-Native Environments," Technical Whitepaper, 2023. [Online]. Available: https://cloudframe.com/resources/ [Accessed: July 2024]

[6] N. Deakin, G. Anglin, and P. Cresswell, IBM CICS Transaction Server for z/OS: Concepts and Facilities, IBM Redbooks, Document SG24-8318-01, 2021. [Online]. Available: https://www.redbooks.ibm.com [Accessed: July 2024]

[7] IBM Corporation, Armonk, NY, USA, "CICS Transaction Server for z/OS Library," IBM Documentation, 2023. [Online]. Available: https://www.ibm.com/docs/en/cics-ts [Accessed: July 2024]

[8] IBM Corporation, Armonk, NY, USA, "VSAM Reference," z/OS Documentation, 2023. [Online]. Available: https://www.ibm.com/docs/en/zos/latest?topic=access-methods-vsam [Accessed: July 2024]

[9] IBM Corporation, Armonk, NY, USA, "DB2 for z/OS Product Library," IBM Documentation, 2023. [Online]. Available: https://www.ibm.com/docs/en/db2-for-zos [Accessed: July 2024]
[10] OpenID Foundation, "OpenID Connect Core 1.0," Nov. 2014. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html [Accessed: July 2024]
[11] D. Hardt, "The OAuth 2.0 Authorization Framework," Internet Engineering Task Force, RFC 6749, Oct. 2012, doi: 10.17487/RFC6749.
[12] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," Internet Engineering Task Force, RFC 7519, May 2015, doi: 10.17487/RFC7519.
[13] N. Sakimura, J. Bradley, and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients," Internet Engineering Task Force, RFC 7636, Sept. 2015, doi: 10.17487/RFC7636.
[14] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," National Institute of Standards and Technology, NIST Special Publication 800-207, Aug. 2020, doi: 10.6028/NIST.SP.800-207.
[15] Apache Software Foundation, "Apache Kafka Documentation," Version 3.7, 2024. [Online]. Available: https://kafka.apache.org/documentation/ [Accessed: July 2024]
[16] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in Proc. NetDB Workshop, Athens, Greece, 2011, pp. 1-7. [Online]. Available: https://notes.stephenholiday.com/Kafka.pdf [Accessed: July 2024]
[17] P. Helland, "Idempotence Is Not a Medical Condition," Commun. ACM, vol. 62, no. 2, pp. 36–44, 2019. [Online]. Available: https://cacm.acm.org/practice/idempotence-is-not-a-medical-condition/ [Accessed: July 2024]
[18] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston, MA, USA: Addison-Wesley Professional, 2003.
[19] V. Vernon, Implementing Domain-Driven Design. Boston, MA, USA: Addison-Wesley Professional, 2013.
[20] The Kubernetes Authors, "Kubernetes Documentation," Version 1.29, 2024. [Online]. Available: https://kubernetes.io/docs [Accessed: July 2024]
[21] Cloud Native Computing Foundation, "Cloud Native Glossary," 2024. [Online]. Available: https://glossary.cncf.io [Accessed: July 2024]
[22] Cloud Native Computing Foundation, "OpenTelemetry Specification," Version 1.33, 2024. [Online]. Available: https://opentelemetry.io/docs [Accessed: July 2024]
[23] Prometheus Authors, "Prometheus Documentation," Version 2.53, 2024. [Online]. Available: https://prometheus.io/docs [Accessed: July 2024]
[24] Grafana Labs, Stockholm, Sweden, "Grafana Documentation," Version 11.1, 2024. [Online]. Available: https://grafana.com/docs [Accessed: July 2024]
[25] HashiCorp, Inc., San Francisco, CA, USA, "HashiCorp Vault Documentation," Version 1.17, 2024. [Online]. Available: https://developer.hashicorp.com/vault/docs [Accessed: July 2024]
[26] Microsoft Corporation, Redmond, WA, USA, "Azure Key Vault Documentation," 2024. [Online]. Available: https://learn.microsoft.com/azure/key-vault [Accessed: July 2024]
[27] Microsoft Corporation, Redmond, WA, USA, "Azure Kubernetes Service Documentation," 2024. [Online]. Available: https://learn.microsoft.com/azure/aks [Accessed: July 2024]
[28] Red Hat, Inc., Raleigh, NC, USA, "Debezium: Stream changes from your database," Debezium Project Documentation, Version 2.7, 2024. [Online]. Available: https://debezium.io/documentation/ [Accessed: July 2024]
[29] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2021.
[30] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston, MA, USA: Addison-Wesley Professional, 2010.