MSW MANAGEMENT -Multidisciplinary, Scientific Work and Management Journal
ISSN: 1053-7899
Vol. 36 Issue 1s, 2026, Pages: 1222-1227

Architectural Refactoring Strategies for Legacy Trade Systems in Cloud-Native Ecosystems

Rang Ganesh Singh Alampur
Jawaharlal Nehru Technological University, Anantapur, Andhra Pradesh, India
rangganeshsingh.alampur@gmail.com

Abstract
Legacy trade systems continue to underpin critical financial operations, yet their monolithic architectures, rigid deployment models, and accumulated technical debt significantly constrain scalability, resilience, and adaptability in cloud-native environments. Direct migration approaches such as lift-and-shift or full system rewrites often fail to address these structural limitations and introduce unacceptable operational and regulatory risks. This paper examines architectural refactoring as a strategic pathway for modernizing legacy trade systems within cloud-native ecosystems. It proposes a layered refactoring framework encompassing structural decomposition, data decoupling, communication transformation, and infrastructure modernization, while preserving deterministic latency, transactional integrity, and regulatory compliance. The study further analyzes domain-specific refactoring patterns and governance considerations essential for trade systems, highlighting trade-offs between cloud elasticity and execution predictability. By framing refactoring as an incremental, risk-aware transformation rather than a purely technical exercise, this work contributes a structured approach to achieving sustainable modernization of trade platforms without disrupting mission-critical operations.

Keywords: Legacy Trade Systems, Cloud-Native Architecture, Architectural Refactoring, Financial Systems Modernization, Distributed Systems Governance

I. INTRODUCTION

Trade systems form the working foundation of existing financial marketplaces in which high-priced trade is conducted, cleared, and settled under stringent constraints on latency, consistency, availability, and regulatory provisions. Traditionally, these systems were developed as monolithic, tightly coupled applications that were deterministically performant on proprietary hardware and on-premises infrastructure [1]. Despite the benefits of this architectural paradigm, such as reliability and predictability, it has also created fixed dependencies across the application logic, data layer, and infrastructure, with long-term technical debt consequences [2]. At the same time, cloud-native computing has also influenced enterprise system design, making it more elastic, fault-isolated, high-speed-deployable, and service-composable through microservices, containers, and event-driven design [3]. A special place in this transformation picture, however, is played by trade systems: they can only offset the dynamism of clouds with the impossibility of implementing the determinism of audits and risk containment. This strain renders architectural refactoring less of a technical undertaking and more of a strategy-based one that necessitates rethinking system limits, the meaning of communication, and state handling [4]. The gradual restructuring of the basic architectural elements that are not interruptive of the vital behavior of the mission is possible through refactoring. Another option to wholesale replacement or sub-surface migration is refactoring. Architectural refactoring in this respect is a modernization mechanism and a governance tool that defines how the past trade platforms can be changed without interference with the market functions [5]. The idea of the sustainable modernization plans in the cloud-native ecosystems is based on the knowledge of this two-fold stance.The industry is now being modernized with two major poles of lift-and-shift migrations, a simple migration of monolithic systems to cloud infrastructure that does not necessarily change the architecture, and greenfield rewrites, which aim to replace the legacy platforms with new ones [6]. Empirical studies have shown that lift-and-shift strategies tend to reproduce the same inefficiencies and extend latency, non-predictability, and operational complexity, while complete rewrites introduce long-term parallel operations, additional regulatory risk, and increased failure risk [7]. The academic foundation for cloud migration and microservice conversion is growing, but it is predominantly based on enterprise systems and consumer-facing applications, with little insight into latency-sensitive, regulation-intensive trade environments [8]. Moreover, no visible domain-specific frameworks have been identified that combine refactoring decisions with trade execution, market integrity, and compliance requirements [9]. As such, there is no methodological approach that practitioners take in the sequence of refactoring projects, architectural trade-offs, and achievement of modernization besides the generic cloud metrics. The specified gap indicates the need for specialized research on architectural refactoring methods tailored to trade systems assumed to be legacy and operating in cloud-native contexts.

The gap in this paper is addressed by viewing architectural refactoring as one of the modernization approaches of migrating legacy trade systems to cloud-native ecosystems. It is also restricted to architectural-level changes, without accounting for low-level code refactoring or vendor-specific implementation details. It also focuses on refactoring policies that ensure deterministic performance, transactional integrity, and regulatory compliance, while allowing scalability, resilience, and operational flexibility. Rather than proposing a generalized model of migration, the paper generalizes architectural principles, refactoring patterns and governance considerations learnt by trade systems [10]. Specifically, this study will have the following objectives:

(1) to explain the architectural constraints which are specific to the past trade systems;
(2) to analyze the weaknesses of current cloud migration plans in the case of trading situations;
(3) to propose a framework for a refactoring strategy of organization within the framework of the principles of cloud-native;
(4) identify domain-specific refactoring patterns and mitigation of risks;
(5) to elaborate on the evaluation principles of gauging outcomes of refactoring in the trade environments.

In this regard, the paper will aim to contribute to a gradual, risk-sensitive strategy for modernizing trade platforms without compromising their core business operations.

II. Characteristics of Legacy Trade Systems

The old trade systems are decades old and have evolved and been modified by market changes, regulatory requirements, and incremental performance needs over the years. These architectures tend to be monolithic, with tightly integrated business logic, data storage, and transaction processing layers tuned for low latency on a limited on-premises system [11]. Flexibility is pushed to the back burner to deterministic behavior which leads to a synchronous communication model, databases of data and centralized control flow. State management is often tightly coupled with application logic, making horizontal scaling and fault isolation particularly challenging. Based on data, old trade systems tend to be built on relational databases, designed to guarantee high data consistency and atomic transactions, which in turn presuppose high trade execution and settlement correctness [12]. The regulatory changes that have been in place over the years, such as reporting requirements, risk controls, and audit requirements, have also added to existing architectures rather than being designed wholesale, which erodes architectures and creates technical debt. Regarding deployment, cycles are often short and extremely risky, as any minor change can spread across closely connected elements. Monitoring and observability are typically retrofitted with minimal real-time knowledge of system behavior when stressed [13]. However, these limitations do not make these systems less resilient or less trusted, as they have proved to be very stable and well aligned with the institutions' processes. This is the robustness versus rigidity duality in that the architecture of the trade system in the modern era can hardly

MSW MANAGEMENT -Multidisciplinary, Scientific Work and Management Journal
ISSN: 1053-7899
Vol. 36 Issue 1s, 2026, Pages: 1222-1227

be disassembled or replaced without breaking the market functioning, and the architectural components of this traditional system are closely related to each other through the order management component, the pricing engines component, the risk checks component, and the settlement modules component and are all operating in a synchronous manner with a shared database [14]. An example of a traditional monolithic architecture of the trade system is shown in Figure 1: the order management component, the pricing engines component, the risk checks component, and the settlement

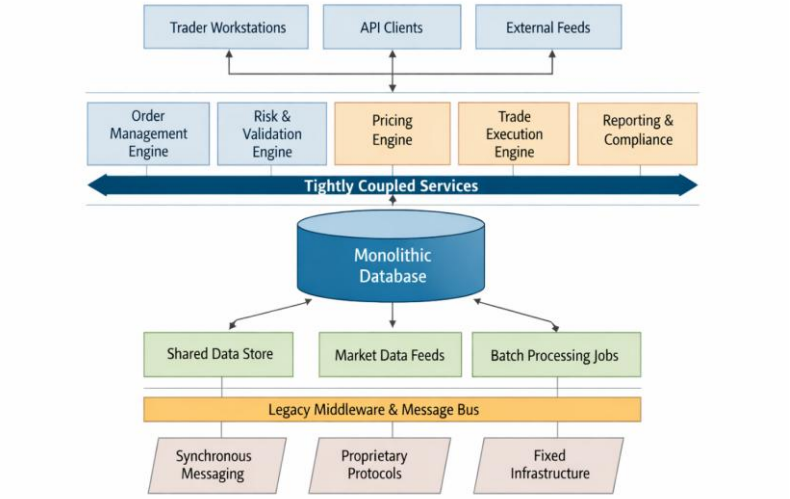

Figure 1. Conceptual Architecture of a Legacy Trade System

Such architectural characteristics have significant consequences for modernization. High integration of the parts amplifies changes, so that any change in a single module would require a ton of regression testing across the whole system. The elasticity of scale, or graceful degradation, properties of the system are also compromised by state- and synchronous-workflow-based systems when exposed to peak market conditions [15]. Also, hardware-based performance optimization can be ineffective in a virtualized or containerized system, and it causes the latency variation and inefficient utilization of resources. The current governance approaches for legacy systems, such as hard-coded compliance checks and audit logic, also complicate transformation, as they cannot be separated into modular services. This leads to a degree of architectural inertia in legacy trade systems: they are functionally sound but structurally inflexible. The discovery of these qualities is a precondition for any refactoring initiative, since it may help architects distinguish between aspects to be preserved to maintain operational integrity and those that can be restructured over time. Lack of such an understanding implies that modernization activities will either destabilize the systems or bring in superficial cloud adoption that has little strategic intent.

III. CLOUD-NATIVE ECOSYSTEM REQUIREMENTS

The desired state for updating legacy trade systems in a cloud-native ecosystem is elasticity and maximum distribution, rather than a sensible trade-off between adaptability and determinism. In this regard, the cloud-native architecture is a framework that enables modularity, resilience, and operational agility without affecting the performance and governance requirements of the trading environment. At its simplest, cloud-native architecture concerns the following concepts: service decomposition, containerization, automated orchestration, infrastructure abstraction, and observability-by-design [16]. The principles also aim to decouple the underlying infrastructure from the application logic, enabling systems to scale, recover, and evolve independently. The process of cloud-native adoption for the trade systems involved in the case under consideration, however, should be discriminating and intentional. It is implemented using microservices to decouple business capabilities for order validation, risk checking, and reporting, as well as for event-based communication, reducing tight coupling. Standardized deployment and controlled scalability, repetition, and auditability can be achieved through containers and declarative infrastructure-based orchestration platforms. Interestingly, the continuous releases and self-testing of cloud-native architecture are also moving to the forefront, offering the opportunity to implement changes incrementally rather than through disruptive release cycles. The target ecosystem can be regarded as a hybrid architectural position, i.e. the application of the concepts of cloud-native to create resiliency and flexibility, however, in a manner such that the latter is not applied in excess in order to guarantee predictability of execution, consistency guarantees and regulatory transparency of trade systems [17].

Table 1. Non-Functional Requirements Specific to Trade Systems in Cloud-Native Ecosystems

Non-Functional Requirement	Description	Implications for Cloud-Native Design
Deterministic Latency	Predictable and bounded execution time for trades	Limits aggressive auto-scaling and asynchronous processing
High Availability	Continuous operation during market hours	Requires fault isolation without state inconsistency
Strong Consistency	Accurate trade execution and settlement	Constrains eventual consistency models
Regulatory Compliance	Auditability, traceability, and reporting	Necessitates immutable logs and governed data flows
Fault Isolation	Localized failure containment	Encourages service-level boundaries and bulkheads
Observability	Real-time monitoring and traceability	Requires end-to-end tracing and structured logging
Security	Protection of sensitive financial data	Demands encryption, access control, and data residency
Change Safety	Low-risk deployment of updates	Favors incremental releases and rollback mechanisms

Cloud-native ecosystems offer advantages but also introduce architectural conflicts, which are inevitable in their use in trade systems. The properties of the cloud-native architecture that do not necessarily align with the deterministic execution model for the reliable trading of resources to guarantee fairness are elastic scaling, dynamic resource allocation, and asynchronous processing [18]. The ability to make informed decisions about scaling depending on the changing workload may introduce differences in the latency and network virtualization, and both multi-tenant infrastructure and network virtualization may negatively affect predictable performance. Other loosely coupled event-driven architectures are also more resilient, but achieving end-to-end transaction visibility and timing guarantees is difficult [19]. All these tensions require architectural trade-offs, such as selective constrained elasticity, tying key services to specialized resources, and introducing hybrid deployment patterns to isolate cloud-native components with performance-sensitive on-premises infrastructure. Trade systems clouds do not always go with the maximization of the abstraction motive, but the domination of architectures. These strains are issues to be identified and managed to support refactoring plans that enable deriving the benefits of the cloud, without compromising the underlying operational integrity of the trading platforms.

IV. ARCHITECTURAL REFACTORING AS A STRATEGIC APPROACH

Modernisation Architectural refactoring is a risk-averse approach to modernisation, the opposite of infrastructure-based migration and wholesale system replacement. Having a traditional trade architecture, business logic, and performance control and optimization are closely

MSW MANAGEMENT -Multidisciplinary, Scientific Work and Management Journal
ISSN: 1053-7899
Vol. 36 Issue 1s, 2026, Pages: 1222-1227

interwoven with architectural decisions that result in the long-term operation life cycles. Consequently, modernization should not be viewed as a compulsory technical complement, but should be regarded as a system framework, a change of its rules, and the job of the operations [20]. Architectural refactoring, selective restructuring of architectural units; e.g. component boundaries, data ownership, communication mechanisms and deployment models, but not the system-observable behavior, is one strategy to deal with this problem. This plays an essential role in trading, where a minor functional dislocation can have a significant financial and regulatory impact. Refactoring is also conceived to reduce the roots of technical debt and structural coupling that make them unable to scale and resilient in the case of lift-and-shift migrations, wherein the processes largely consist of the migration of the existing systems into the cloud resources and do not require a change in the rigidity of the architecture [21]. Meanwhile, it also does not entail the higher risks of wholesale system rewrites, such as periods of coexistence between two systems, loss of domain knowledge, and burdensome re-certification under regulation. Architectural refactoring can be used to transform incremental change into a reversible and controlled set of steps so that the organization can go on to modernize as compared to disruptive transformation programs. This type of incrementalism is concordant with the feasibility of the systems of trade, which should be in place, audited and performant during the life of the modernization process.Moreover, the architectural refactoring is a management instrument. Refactoring results in greater architectural visibility and decision traceability by establishing clear architectural boundaries, standard interfaces, and visible execution paths [22, 23]. This kind of transparency encourages adherence to the regulation, operational risk, and proximate system custodianship. Refactoring reinstates architecture will and governance in cloud-native ecosystems that might be covered under abstraction layers to obscure system behavior. Then, architectural refactoring is nonexistent as the tradeoff between competing strategies of coping with the modernization but a strategic discipline as a tradeoff between the imperatives of the cloud-native domain and the determinism, responsibility and stability that trade systems expect [24, 25].

Table 2. Comparative Analysis of Modernization Approaches for Legacy Trade Systems

Dimension	Lift-and-Shift Migration	Full System Rewrite	Architectural Refactoring
Primary Objective	Infrastructure modernization	Architectural replacement	Structural evolution
Scope of Change	Deployment environment only	Entire application stack	Targeted architectural layers
Change Granularity	Coarse-grained	Coarse-grained	Fine-grained and incremental
Impact on Core Logic	None	Complete reimplementation	Preserved with selective restructuring
Latency Predictability	Often degraded	Uncertain during transition	Preserved and optimized incrementally
Regulatory Risk	Low initially, increases over time	High due to revalidation needs	Controlled through staged compliance checks
Operational Continuity	High	Low during transition	High
Technical Debt Reduction	Minimal	High but delayed	Progressive and measurable
Time to Realize Benefits	Short-term	Long-term	Continuous
Cost Profile	Low upfront, high long-term	Very high upfront	Distributed and manageable
Knowledge Retention	High	Low to moderate	High
Suitability for Trade Systems	Limited	Risk-prone	Highly suitable

The longer analogy provides the architectural refactoring with the context-sensitive modernization of the legacy trade systems that will be put in the environment of cloud-native ecosystems. Lift-and-shift migration is fast, but it carries permanently entrenched structural problems, is generally more costly over the long term, and reduces performance predictability. Direct rewrites of systems, as attractive as they are within the lens of the architecture, have too much operational, financial and regulatory risk in a system where the continuity and stability of the system cannot be jeopardized. Architectural refactoring takes up an intermediate position between the two extremes of change, facilitating fine-grained reversible change, which allows the modernization to proceed in a manner that is consistent with the organizational risk tolerance. That cloud-native enables one to gain powers such as improved resilience and observability, without reducing determinism or compliance, due to its capacity to maintain underlying logic and to change the shape of systems simultaneously. One should also mention that refactoring converts modernization into a systematic architectural potential rather than a project that ends, and it facilitates the market, regulatory and technological change responding process. The position of strategy in the systems of trade is not only good, but preconditions of the sustainable evolution.

V. REFACTORING STRATEGY FRAMEWORK

The legacy systems may be architecturally refactored and require a methodical approach to handling deep-seated coupling, performance sensitivity, and compliance constraints, while allowing a gradual, systematic migration of the old systems to new cloud-native systems. Single or informal refactoring, e.g., dividing services without considering data ownership, adding containers without examining the meaning of the communication, etc., is unlikely to yield a sustainable architectural change. The refactoring strategy framework presented below aims to address this weakness by providing a layer- and architecture-based framework that explains what to refactor, why, and how to transform the system in a risk-free, gradual manner.The framework is driven by three major observations. First, the legacy systems also entrench the architecture choices made on numerous levels, including structure, data, communication and infrastructure, that cannot be optimized separately to cause instability. Second, most of the mission-critical systems depend on the maintenance of deterministic behavior and continuity of operation, and they cannot be replaced on a large scale. Thirdly, the benefits of the cloud-native adoption will become a reality as the responsibilities of an architecture are defined and executed. In this sense, the framework contains a decomposition-by-concern philosophy, which allows the refactoring to be carried out on a layer-by-layer basis, without causing an effect of destabilization to the system. This model is written in an abstract form and consists of four dependent dimensions of refactoring: structural, data, communication, and infrastructure. The architectural constraint dimensions are set to different constraint classes and clearly defined to ensure that the spread of change is not propagated too far. Incremental development, reversibility, and ongoing validation are the focal elements of the model and are useful in ensuring that organizations abreast with the pace of modernization as well as assuming risks. It is important to note that it considers refactoring to be an endless cycle of architectural governance and not a migration project. The framework enables refactoring activities to be sequenced and communication between activities to be directly controlled, providing a repeatable blueprint for transforming legacy systems into a modern system without interfering with the integrity and compliance requirements of operations.

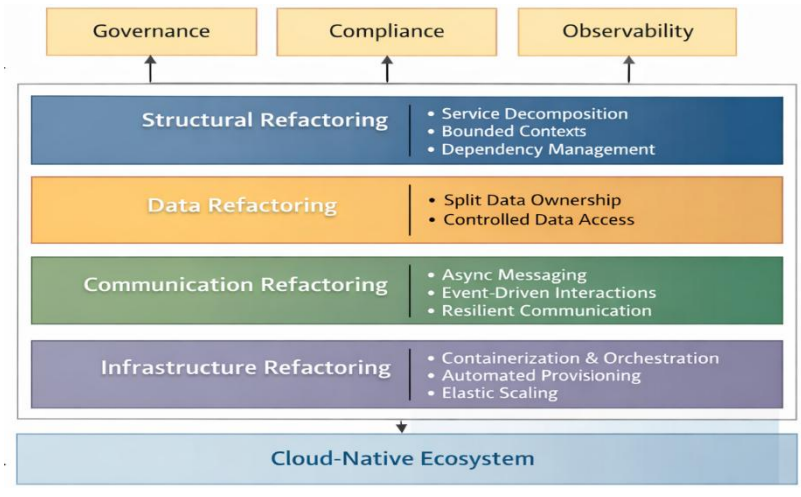

Figure 2. Layered Refactoring Strategy Framework for Legacy Trade Systems

5.1 Structural Refactoring

Structural refactoring is a push to simplify the current trade system to elucidate its architectural advantages and dependency relationships. Within a traditional trade system, with obsolete added features and regulatory extensions, the functional units, e.g., order capture, pricing, risk validation, execution, and reporting, are woven into monolithic forms. This close level of interconnection amplifies variations, increases the separation of failures, and also restricts autonomous development. Structural refactoring attempts to undo this erosion by making the extent of limited contexts, which represent business capabilities, explicit and imposing directional constraints between the abilities of business capabilities. Structural refactoring is intended to achieve architectural clarity rather than the functional decomposition

MSW MANAGEMENT -Multidisciplinary, Scientific Work and Management Journal
ISSN: 1053-7899
Vol. 36 Issue 1s, 2026, Pages: 1222-1227

itself. The components are developed, and legacy core logic is separated with the assistance of such techniques as the modularization, facade services, and anti-corruption layers. This further enables the new cloud-native services to interact with the legacy functionality; however, they should not assume its constraints. It is also worth mentioning that a structural refactoring has no changes in the current business semantics and execution paths and therefore does not change any trade behavior but simply redistributes the architectural responsibilities. It does not entail the rewriting of algorithms, execution code, or regulatory code. Rather, it is preoccupied with developing a homogeneous architecture that can be upgraded over time. Structural refactoring can reduce systemic risk by defining ownership boundaries, isolating volatile components, and preconditioning data, communication, and infrastructure. This is a highly sensitive area as far as the trade systems are concerned, when it comes to the necessity of creating a balance between the progress in modernization and the continuation of operations.

5.2 Data Refactoring

The centralized and shared persistence models can be used as a marker to identify the objective of the data refactoring, which is one of the most difficult limitations of the legacy trade systems. Traditionally, trade platforms use monolithic relational databases that provide high consistency across functional domains. This is an auditable, transactionally correct design with significant coupling between the application's components (through shared schemas) and would not be feasible to deploy and scale the independent components. Data refactoring: in order to separate the ownership of data and maintain the integrity of data that is required for financial operations. Each refactored unit is assigned a responsibility for its own data, with clear read and write boundaries. The methods of decomposing databases, isolating the schema, and controlled replication are likewise suggested gradually to avoid disruption. The trade systems should be arranged in sequence to ensure that there is no loss of trade guarantees, reconciliation procedures and audit trails. Read-write separation, Event-based data propagation, such as languages in which a selective relaxation of coupling can be permitted, but not the unsafe models of eventual consistency. Refactoring of data is limited only to the boundaries of data models, access paths, and transaction limits, but not beyond financial semantics or even the reformulation of legal logic. Critical paths of execution may also be highly consistent, and replicated or derived data may be used in other non-time-sovereign functions (e.g. reporting or analytics). Scalability, stability and accuracy. Given that competition is removed from shared data stores and ownership is assured, data refactoring enables such scalability. It is essential in cloud-native ecosystems, which require altering the distributed architecture to meet the high data-integrity requirements of trade systems.

5.3 Communication Refactoring

Communication refactoring is the process of improving the communication pattern between components to be more efficient and less prone to failures. The ancient conservative systems of trade are largely anchored in request-response communication patterns that are not only synchronous but also tightly coupled service calls, which reveal inflexible patterns of advancement. Although this type of strategy makes workflows deterministic, it is far more responsive to latency and can lead to cascading failures during times of stress. Refactoring communication adds more valid models of communication without degrading traceability or execution assurance. In refactoring the communication, the focus is therefore not on eradicating synchronous forms of communication but on applying determinism selectively to those areas where it should be applied. Non-critical or downstream processes (as mentioned), such as post-trade reporting, compliance, and analytics, are also added to asynchronous message-based workflows, event-driven buffering, and queue-driven buffering. This eliminates lock-up dependencies and enables the critical execution paths to be load-balanced. Such patterns as publish-subscribe, message queues, and event streams can also be decoupled using the protocol and semantics of messaging, failure management, and retries without losing the audible event streams. Specifically, it does not permit modifications to the business logic or the principles of performance. Observability is considered a special case because, in asynchronous systems, the execution flow cannot be automatically observed. Trade systems Trade systems Trade systems Communication refactoring of trade systems is an instrument that has been employed to achieve fault tolerance and scalability, requiring controlled execution time and regulatory control.

5.4 Infrastructure Refactoring

The final reference for the refactoring strategy framework is infrastructure refactoring, which is the model for the deployment, execution, and operation of the trade systems. The legacy platforms will be associated with fixed, hardware-intensive systems that perform very well in terms of consistency but poorly in terms of flexibility. Projects of infrastructure refactoring, such as migrating to cloud-native models through containerization, orchestration, and automated provisioning, without up-front costs. Contrary to lift-and-shift migrations, infrastructure refactoring is gradual, where the move is made when enough structural and communication decoupling has been realized. This will ensure that the infrastructure's flexibility does not increase the faults in the architecture. Standardization of runtime environment, containerization, and orchestration platforms, and runtime environment standardization are the sources of health and recovery deployment and management. This concept of elasticity is selectively applied to trade systems, which attach latency-sensitive components to resources dedicated to them, and construct peripheral services on top of dynamic scaling. Scope Infrastructure Refactoring Scope entails deployments, operational tooling automation, configuration management and runtime isolation. It is not associated with modifications to application logic or execution semantics. Making infrastructure a better place to observe and govern is important because it enables similar, consistent monitoring, documentation, and policy execution across environments. The discipline can help build resilience and sustain cloud-native ecosystems without compromising determinism. With the help of trade systems, architecture transformation is also achieved through infrastructure refactoring, which aligns the operations' capabilities with the constraints and responsibilities defined in earlier layers of the refactoring.

VI. Refactoring Patterns for Trade Systems

Refactoring patterns can be viewed as a set of concepts that specify where and how architectural changes should be made. The refactoring strategy framework identifies the main points of change in the architecture, and these concepts are applied through the particular patterns peculiar to the trade systems. Refactoring patterns bring on board structural and interaction patterns that are working in the legacy platform and therefore provide the option to improve the platform in small steps without interfering with execution integrity, regulatory issues, or system stability. The patterns of trade environment should consider deterministic latency, data accuracy rigidity, and determined availability and not enterprise-based templates of modernization. Wrapping legacy behavior in stable interfaces, which are patterns that allow selective modernisation, are facade-based isolation, incremental service extraction and event-driven augmentation. It is worth noting that the patterns support reversibility and coexistence; thus, the refactored elements can be used alongside the legacy modules in the long term. This co-existence is also the core of the trade systems, where failing to work a system and/or failing to work long-term is a normal occurrence due to regulatory approval and operational risk limits. Moreover, one of the governance tools to standardize the architectural decision-making and minimize ad hoc changes is the refactoring pattern. They would allow certainty to be pruned when done in a systematized manner, would allow making of trade-offs to be made systematically, and would impose on intuitional aversion to risk the modernization efforts being done to date in a systematic manner. Table 3 summarises the main refactoring patterns of the trade systems and categorizes them by purpose, scope, and anticipated benefits, which can be used by architects as a helpful toolkit for a controlled, cloud-based modernisation.

MSW MANAGEMENT -Multidisciplinary, Scientific Work and Management Journal
ISSN: 1053-7899
Vol. 36 Issue 1s, 2026, Pages: 1222-1227

Table 3. Refactoring Patterns Applicable to Legacy Trade Systems

Refactoring Pattern	Description	Primary Use Case	Key Benefits	Trade-System Considerations
Strangler Pattern	Incrementally replaces legacy components by routing functionality to new services	Progressive modernization	Low risk, reversible change	Requires careful traffic routing
Façade Pattern	Wraps legacy systems with stable interfaces	Legacy isolation	Reduces coupling, improves control	Latency overhead must be managed
Anti-Corruption Layer	Prevents legacy models from contaminating new services	Domain protection	Preserves clean service boundaries	Additional translation logic
Event-Driven Augmentation	Introduces event streams alongside synchronous flows	Scalability and resilience	Decoupling, asynchronous processing	Observability and ordering guarantees
Sidecar Pattern	Externalizes cross-cutting concerns (logging, compliance)	Governance and observability	Separation of concerns	Resource overhead
Bulkhead Pattern	Isolates critical services to prevent cascading failures	Fault containment	Improved resilience	Resource partitioning complexity
Circuit Breaker	Prevents failure propagation under stress	Runtime stability	Graceful degradation	Timeout tuning critical
Legacy Adapter	Translates legacy protocols and data formats	Interoperability	Enables coexistence	Maintenance overhead

VII. CHALLENGES AND LIMITATIONS

The structural reorganization of the traditional trade systems in the form of cloud-native systems is strategically beneficial and constrained by a complex of interdependent problems, which, on the one hand, are embedded in the organization structure and managed by the authorities and, on the other hand, are anchored in technical intricacies. There are also risk-averse trade systems operating in environments where the stability, the precision of system operation, and the ability to abide by it cannot be negotiated. Even the tiniest architectural alteration can reveal certain hidden dependencies or performance sensibilities which cannot be confronted within the traditional operating environment. In contrast to greenfield systems, legacy trade systems may have been in place for decades, with years of assumptions and institutional knowledge that are rarely documented and difficult to restructure. The presence of such architectural inertia makes the evaluation of effects more complex and the refactoring process even more vague. In addition, the inclusion of an abstraction layer is also a feature of cloud-native systems and entails the virtualization, orchestration, and dynamically allocated resources that may conceal the execution behavior and, as a result, the performance may be difficult to predict. However, with this scale and the resulting resilience, such abstractions also diminish the ability to directly manage how systems will behave, which is undesirable in latency-sensitive workloads such as trading. Refactoring should be done in a small margin between the value of architecture and the operational risk. Moreover, the processes of modernization may be of longer periods of co-existence of the old and refactored features, which complicate the system even more and make it even more difficult to operate. In addition to the technical influences, there are organizational and regulatory influences that determine the refactoring strategies' practicality. The existing mechanisms of government, audit mechanisms and mechanisms of regulatory approval do not presuppose a model of gradual, continuous change, and architectural change is imposed on them. Refactoring projects also have financial constraints, business conflicts of interest and failure as limiting factors. Combined with the restrictions mentioned above, it is worth noting that architectural refactoring is not an a priori solution; it is a case-by-case approach whose performance depends on disciplined implementation, unrealistic expectations, and long-term institutional investment. These deficiencies need to be appreciated so that refactoring can be considered as a controlled evolution process as opposed to a justifiable product of modernization.

7.1 Organizational and Cultural Resistance

One of the most perennial non-technical issues in refactoring of the legacy trade systems is the organizational and cultural resistance. Such systems tend to have specialised teams that are highly knowledgeable about the domain and have been operating within the system for a period. Although such a skill is indispensable, it may also lead to opposition to architectural change, especially when refactoring efforts are seen as risking system stability or role-based dependencies. In a system where system outage or poor performance is a serious financial and reputational cost, stakeholders have fewer incentives to pursue long-term flexibility in favor of the short-term reliability of their system. Moreover, refactoring projects often require cross-functional collaboration among development, operations, compliance, and risk management teams. Discussion of incentives and priorities across these categories may slow decision-making and lead to ownership disputes. Continuous delivery and automated testing are also cloud-native practices that will be incompatible with the current governance models built around a low-frequency, high-assurance release model. Architectural refactoring will otherwise be a limited endeavor to surface-level improvements and change without change management and leadership backing.

7.2 Hidden Coupling and Architectural Opacity

Incremental refactoring faces serious challenges due to hidden coupling and an architecturally opaque nature. Legacy trade systems have unrecorded dependencies, common data access points, and unspoken execution rules governing long-term system operation. Such couplings are usually not reflected in architectural documentation and only manifest when refactoring. Consequently, modifications which are supposed to be localized could have unforeseen side effects on the system. Architectural opacity also makes it difficult to study impact and assess risk, as it constrains architects' ability to rank refactoring changes. In trade systems where accuracy and determinism are of paramount importance, the discovery of latent dependencies in the transformation process can cause a delay in modernization or lead to expensive rollbacks. To overcome this difficulty, an initial investment in architectural discovery, observability, and dependency mapping is required, even though it may not provide direct business value, as it will be needed to perform safe refactoring.

7.3 Performance and Latency Trade-offs

A trade system's performance predictability is a feature, and architectural refactoring can introduce trade-offs that affect it. With satisfactory overall performance, it is possible to add layers of abstraction, inter-service communication, and network hops, increasing latency and variance. This variability can impair the execution validity and the credibility of the system in a latency-sensitive trading system. Additionally, to this challenge, the cloud-native platforms introduce multi-tenancy and dynamic resource scheduling that has the potential of introducing non-deterministic performance behavior. It reduces the benefits of elasticity typically associated with the cloud setting, even though the effects can be mitigated through techniques such as resource pinning and priority scheduling. Refactoring, in turn, ought to assess the performance impacts cautiously on a phase-by-phase basis and may need to limit the use of cloud services in sensitive execution streams.

7.4 Regulatory and Compliance Constraints

The architectural refactoring of trade systems subject to regulatory oversight is subject to very strict limitations. A revalidation, recertification, or long audit can be run when the system's structure, execution logic, or data flow changes. These procedures can become an enormous obstacle to the modernization process and the pursuit of experimentation. What is more, compliance logic can be externalized more easily to modular services, but may be hard to audit where execution traces and data lineage have not been carefully polished. Refactoring should then be used as a principle to address compliance issues in architectural design. These involve maintaining a clear audit trail and establishing a trackable execution and compatibility process between the change management process and the regulatory expectations process. It may pose an unacceptable governance risk, outweighing the positive impact of refactoring.

MSW MANAGEMENT -Multidisciplinary, Scientific Work and Management Journal
ISSN: 1053-7899
Vol. 36 Issue 1s, 2026, Pages: 1222-1227

7.5 Economic and Operational Limitations

The quantity and speed of architectural refactoring and its speed are also limited by economic and operational constraints. Long-term coexistence of legacy and refactored parts can lead to incremental transformation and increase operational and maintenance complexity. Parallel architectures require additional tooling, skills, and coordination, which would strain organizational resources. Financial constraints and competing strategic priorities can also limit investments in refactoring projects, which offer long-term payoffs but only short-term payoffs. By this, organizations may focus on the tactical improvement rather than an architectural development that is holistic. Such economic realities serve to emphasize the fact that priority, incremental application, and expression of value need to be given prominence in attaining architectural refactoring in the process of implementing the discussed legacy trade systems.

Conclusion

The review was a synthesis and critical review of current architectural strategies of modernizing the legacy trade systems, specifically understanding the value of architectural refactoring in cloud-native ecosystems. The discussion indicated that legacy trade systems are placed in a separate category of enterprise systems, with severe latency constraints, high consistency guarantees, and extensive regulatory oversight. These properties pose major constraints on the applicability of generic cloud migration strategies commonly used in the literature. As mentioned in the review, the structural rigidity and risk sensitivity of trade systems often cannot be overcome by methods such as lift-and-shift migrations or a complete rewrite. This paper unified the concepts of software architecture, cloud-native design, and financial systems engineering and positioned architectural refactoring as a cohesive approach that bridges the constraints of legacy systems with the current architectural paradigm. The consulted frameworks and patterns all emphasise gradual change, clear architectural lines, and governance-driven evolution. The conceptual model of dealing with complexity that remained operational yet remained manageable through multi-layered refactoring became a common and successful approach to complexity management, including structural, data, communication, and infrastructure dimensions. Other problematic aspects of the review were the recurrent challenges that were reported in the literature and in the practice in the industry, such as hidden architectural coupling, variable performance, regulatory friction and organizational resistance. The restrictions suggest that architectural refactoring is not a prescriptive intervention but rather a situational discipline that requires long-term institutional investment and architectural management. Overall, the review has added a coherent knowledge of how the architectural refactoring has developed as a strategic solution to the modernization dilemma of trade systems. It also outlines that future empirical studies, standard assessment metrics, and models of governance that enable consistent, compliant architectural evolution are needed in cloud-native financial settings.

References

[1] Bass, L. (2012). *Software architecture in practice*. Pearson Education India.
[2] Cunningham, W. (1992). The WyCash portfolio management system. *ACM Sigplan Oops Messenger*, *4*(2), 29-30.
[3] Newman, S. (2021). *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".
[4] Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
[5] Rozanski, N., & Woods, E. (2012). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.
[6] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, *35*(3), 24-35.
[7] Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, *4*(5), 22-32.
[8] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, 195-216.
[9] Subramoni, H., Petrini, F., Agarwal, V., & Pasetto, D. (2010, April). Streaming, low-latency communication in on-line trading systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)* (pp. 1-8). IEEE.
[10] Júnior, A. A., Misra, S., & Soares, M. S. (2019, June). A systematic mapping study on software architectures description based on ISO/IEC/IEEE 42010: 2011. In *International Conference on Computational Science and Its Applications* (pp. 17-30). Cham: Springer International Publishing.
[11] Hohpe, G., & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.
[12] Gray, J., & Reuter, A. (1992). *Transaction processing: concepts and techniques*. Elsevier.
[13] Kim, G., Behr, K., & Spafford, G. (2013). The phoenix project: a novel about IT, devOps, and helping your business.
[14] Bass, L., Weber, I., & Zhu, L. (2015). *DevOps: A software architect's perspective*. Addison-Wesley Professional.
[15] Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.".
[16] Pahl, C. (2015). Containerization and the paas cloud. *IEEE Cloud Computing*, *2*(3), 24-31.
[17] Burns, B., Beda, J., Hightower, K., & Evenson, L. (2022). *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.".
[18] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, *41*(6), 205-220.
[19] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., ... & Zazworka, N. (2010, November). Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (pp. 47-52).
[20] Kruchten, P., & Ozkaya, I. (2019). *Managing technical debt: reducing friction in software development*. Addison-Wesley Professional.
[21] Leymann, C. F. F., Retter, R., Schupeck, W., & Arbitter, P. (2014). Cloud computing patterns. *Springer, Wien. doi*, *10*(2014), 978-3.
[22] Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, *52*(1), 40-44.
[23] Hogan, M., Liu, F., Sokol, A., & Tong, J. (2021). *NIST cloud computing standards roadmap. National Institute of Standards and Technology*.
[24] Humble, J., & Farley, D. (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.
[25] Van Steen, M., & Tanenbaum, A. S. (2017). *Distributed systems* (p. 20). Leiden, The Netherlands: Maarten van Steen.