10.48047/jocaaa.2025.34.12.32

Domain-Driven Design and BIAN Service Models for Financial Microservices

Shashi Kumar Munugoti
Independent Researcher, USA

Abstract

The infrastructural modernization of financial services requires architectural methodologies that effectively balance functional decomposition, service reusability, and regulatory compliance while allowing for rapid innovation. This article examines how Domain-Driven Design principles, integrated with Banking Industry Architecture Network reference models, set up an end-to-end framework for building cloud-native financial microservices. By aligning business domains to standardized service definitions, financial institutions can achieve increased modularity, cross-platform interoperability, and the rapid delivery of customer-facing capabilities. The integration of these methodologies addresses a number of the most important challenges faced during the transformation of enterprises: the definition of service boundaries, the reuse of components, and the governance of architectures in distributed systems. These frameworks have shown practical applications in customer management, payment processing, and lending operations, while meeting the requirements for governance through event-driven architectures and thorough security implementations. The emerging patterns that can be identified, including machine learning integration and an event streaming platform, enable financial institutions to keep on the path of evolution. This synergistic approach forms established routes on how to traverse through the change of monolithic systems to modern architectures, which are more adaptable, resilient, and efficient in operations without going against regulatory and security measures that are crucial to finances.

Keywords: Domain-Driven Design, BIAN Service Domains, Financial Microservices, Cloud-Native Architecture, Service Decomposition

1. Introduction

Financial institutions are facing an increasingly complex technology landscape driven by competitive pressure from digital-first entrants, high expectations from customers for seamless experiences, and severe regulatory requirements across multiple jurisdictions. The challenge of architectural modernization in enterprise systems has been the subject of much research, with empirical studies showing that reference architectures offer great value in guiding the decomposition of complex systems while maintaining conformance between business capability and technical implementation [1]. The old monolithic architectures are predictable but inflexible to organizational responsiveness and prevent the speedy release of new features. The microservices architecture has the distinct benefits that encompass independent scalability, fault isolation, and accelerated development cycles. Nevertheless, unless deployed using stringent decomposition, microservices will lead to distributed monoliths, and redundant functionality and integration complexity will eat away at the benefits expected.

Domain-Driven Design provides a strategic approach to the modeling of software, centered on business domain understanding, while BIAN reference models provide industry-standardized service definitions tailored for banking operations. The BIAN Service Landscape is a comprehensive framework that organizes banking capabilities into distinct service domains spanning core operational areas, customer engagement functions, risk and compliance operations, and supporting infrastructure services [2]. Putting

10.48047/jocaaa.2025.34.12.32

these complementary frameworks together lays a robust foundation for financial microservices to be both business-aligned and architecturally sound. It addresses key questions in service-oriented architecture, such as how to determine appropriate service boundaries, what the right level of granularity is for optimal reuse, and how consistency is maintained across a distributed development team.

The financial services industry faces architectural challenges unlike many other industries: The need for transactions to be absolutely consistent and auditable; regulatory reporting demands data lineage; the security requirements go way beyond what would normally be found in any average enterprise. Equally, due to the interconnected nature of most financial operations, services need to not only collaborate across complex workflows but also be loosely coupled. These constraints render the structured approaches afforded by DDD and BIAN especially useful in providing proven patterns that manage complexity while preserving system integrity.

2. Background and Foundational Concepts

2.1 Domain-Driven Design Principles

Domain-Driven Design is a systematic approach to complex software development, putting much stress on the importance of creating software models that reflect a deep understanding of business domains. One key aspect of this methodology is the collaboration between domain experts and development teams in establishing a ubiquitous language: terminology that is kept consistent across business discussions, code implementation, and system documentation. This helps minimize miscommunication and ensures that the technical implementation indeed reflects business intent. Research in microservices architectural patterns has demonstrated that proper service decomposition based on domain boundaries significantly reduces coupling while improving system maintainability and the ability to evolve [3].

At the core of DDD lies the concept of bounded contexts, which establish very clear limits around the capabilities of business. Every bounded context has its domain model, shields its implementation, and interacts with other contexts via clearly defined interfaces. This method will not build overly complex and tightly coupled domain models that attempt to address all uses simultaneously. In financial systems, bounded contexts may separate customer relationship management from transaction processing, or lending operations from deposit services, letting each area develop independently with clear integration points. DDD also introduces a set of strategic design patterns, which include entities, value objects, aggregates, and domain events. Entities have unique identities that survive beyond their operational lifetime, such as customer accounts or loan applications. Value objects describe aspects without conceptual identity, such as monetary amounts or addresses. Aggregates assemble related entities and value objects under a consistency boundary, ensuring that business rules are executed atomically. Domain events represent significant business occurrences and enable reactive architectures and temporal coupling reduction between services.

2.2 BIAN Service Architecture Framework

Banking Industry Architecture Network developed a general reference architecture for the banking industry, catering to the requirement for standardization in an industry where multiple players use similar functionalities with different implementations. According to BIAN, service domains define distinct business capabilities in banking operations. Each of these capabilities encapsulates certain functionalities that are accessed through well-defined interfaces. The BIAN architecture organizes functional patterns in business areas comprising operations, sales and service, customer management, products and services, and support functions. The proposed architecture comprehensively covers all the banking capabilities by providing standardized service domain definitions [4]. These include anything from directly customer-

10.48047/jocaaa.2025.34.12.32

facing capabilities, like account management, to internal back-office capabilities, such as regulatory compliance reporting. This framework structures banking capabilities into service domains, service operations, and control records, respectively, in a hierarchical form. Service domains reflect cohesive business capabilities; examples include customer offer management, payment execution, and credit risk assessment. Each service domain provides standardized service operations that define how an external party interacts with the capability. The control records are the primary business objects maintained in each service domain to handle the state and coordinate behavior across the operations.

DDD Pattern	Financial Service Domain	Primary Function	Integration Complexity	Reusability Factor
Bounded Context	Customer Relationship Management	Customer profile isolation	Medium	High
	Transaction Processing	Payment authorization	High	Medium
	Lending Operations	Loan origination and servicing	High	Medium
	Deposit Services	Account management	Medium	High
Aggregate	Account Entity	Balance calculations and transaction history	Low	High
	Loan Application	Credit assessment and approval workflow	Medium	Medium
Value Object	Monetary Amount	Currency representation	Low	Very High
	Address	Location data standardization	Low	Very High
Domain Event	Transaction Completed	Reactive architecture trigger	Low	High
	Account Updated	State change notification	Low	High

Table 1: Domain-Driven Design Pattern Implementation in Financial Microservice [3, 4]

3. Integrating DDD and BIAN for Financial Microservices

3.1 Strategic Service Decomposition

Such a decomposition of monolithic financial systems into microservices requires an in-depth analysis aimed at finding suitable service boundaries with a view to having them aligned with business capabilities while simultaneously providing technical independence. For this decomposition, Domain-Driven Design's bounded contexts provide the conceptual basis needed to establish logical boundaries around cohesive functionality. Applied to financial services, bounded contexts can demarcate core banking operations,

10.48047/jocaaa.2025.34.12.32

customer relationship management, product catalog management, and financial reporting as separate domains that interact through explicit interfaces. BIAN service domains enhance DDD bounded contexts by providing tried-and-tested definitions of industry capabilities in banking. Rather than base service boundary identification exclusively on organizational analysis, architects can refer to BIAN's directory of service domains as a starting point, adapting these standard definitions to meet institutional needs. In this way, decisions on decomposition are accelerated while alignment with industry best practices and regulatory expectations is assured.

The mapping between DDD contexts and BIAN domains is not always one-to-one. One BIAN service domain may cover several bounded contexts, for example, when institutional complexity justifies finer-grained separation. On the other hand, for small-scale implementations, closely related BIAN domains can be combined into a single bounded context. Studies on challenges related to microservices granularity have shown that architecture metamodeling techniques that take business capabilities, data cohesion, and operation characteristics into account offer structured approaches for deriving the correct boundaries of services in a complex enterprise system [5]. A financial transaction process, for example, may segregate authorization, settlement, and reconciliation into different contexts, each of which would map to an appropriate BIAN service domain.

3.2 Implementation Design Considerations

Within each microservice, DDD tactical patterns guide internal structure and behavior. Aggregates define transactional consistency boundaries, ensuring that business invariants are maintained through all state transitions. In financial services, an account aggregate might include the account entity along with transaction history, balance calculations, and associated value objects representing currency and interest rates. The BIAN service landscape organizes capabilities through service domain classifications that discriminate between operational management, directional governance, and administrative support functions, providing clear guidance for structuring microservices implementations aligned with banking business patterns [6]. All changes to the account come through the aggregate root, which enforces business rules like minimum balance or maximum transaction amounts. Domain services implement business logic that spans multiple aggregates or doesn't naturally belong to any one entity. Risk assessment might require data from customer profiles, account balances, transaction history, and market conditions.

Business Area	Service Domain	Domain Classification	Primary Operations	Cross-Domain Dependencies	Standardization Level
Operations	Payment Execution	Operational Management	Initiate, Update, Execute	Account Management, Fraud Detection	High
Sales and Service	Customer Offer Management	Directional Governance	Initiate, Evaluate, Activate	Customer Profile, Product Catalog	High
Customer Management	Customer Profile Management	Administrative Support	Create, Update,	Authentication,	Very High

10.48047/jocaaa.2025.34.12.32

			Retrieve	Reference Data	
Products and Services	Account Management	Operational Management	Open, Update, Close, Retrieve	Transaction Processing, Reporting	High
Support Functions	Reference Data Management	Administrative Support	Define, Update, Retrieve, Exchange	All Service Domains	Very High
Risk Management	Credit Risk Assessment	Directional Governance	Evaluate, Update, Report	Customer Profile, Account Data	Medium
Compliance	Regulatory Compliance Reporting	Administrative Support	Collect, Aggregate, Submit	All Transactiona l Domains	High
Operations	Transaction Processing	Operational Management	Authorize, Settle, Reconcile	Payment Execution, Account Management	High

Table 2: BIAN Service Domain Classification and Operational Characteristics [5, 6]

4. Enabling Service Reusability

4.1 Shared Service Capabilities

Financial organizations often have redundant functionality across applications, where customer identification, address validation, or transaction validation logic is reproduced in many systems. Microservices architecture combined with BIAN service domain definitions provides opportunities to eliminate this redundancy through shared-service capabilities. Customer profile management can be implemented once as a dedicated microservice, consumed by all applications requiring customer data, ensuring consistency and reducing development effort. Candidates for shared services can be identified by analyzing cross-functional capabilities that crop up across multiple business processes. Authentication and authorization fit this pattern naturally, since practically every financial service requires the identity of users or customers to be verified, and their access controlled. Likewise, reference data management-currency codes, country codes, product catalogs, and fee schedules are a natural fit for centralization, ensuring all systems work from consistent definitions. Notification services handling email, SMS, and push notifications are another common shared capability, abstracting the channel of communication away from business logic.

Any shared service implementation needs to balance reusability carefully against coupling. A service that is sufficiently generic to handle all conceivable uses tends to become complex, hard to change, and difficult to test. Research into microservices architectural styles has shown that successful service decomposition strategies need to consider multiple factors, such as business capability alignment, data ownership boundaries, and operational independence, in order to maximize reusability while avoiding undesirable coupling [7]. Conversely, services optimized for particular consumers might not be the best

10.48047/jocaaa.2025.34.12.32

fit for other usage patterns. The bounded context concept attempts to handle this tension by acknowledging that different contexts are likely to require a different perspective on shared concepts.

4.2 Workflow Coordination Patterns

Complex financial workflows often span multiple service domains and thus require coordination across microservices while maintaining loose coupling. Processing such as payment may be done through customer authentication, account validation, fraud detection, execution of transactions, delivery of notifications, and accounting updates. Orchestration patterns establish explicit coordination logic that manages this complexity, typically implemented through workflow engines or choreography through domain events. The BIAN service operations framework defines standardized interaction patterns, including initiate, update, retrieve, and exchange operations that govern how services coordinate during business process execution, providing consistent mechanisms for service composition across financial workflows [8]. Orchestration concentrates the workflow logic in a special coordinator that calls services in a sequence and manages errors by compensating transactions in case of failure. This not only gives a clear view of the process end-to-end, but also simplifies the execution of complicated branching logic.

Shared Service Capability	Implementation Frequency	Consumer Services	Coupling Level	Deployment Complexity	Maintenance Overhead
Customer Profile Management	Single Instance	All Customer-Facing Services	Low	Low	Low
Authentication and Authorization	Single Instance	All Services	Very Low	Medium	Low
Reference Data Management	Single Instance	All Services	Very Low	Low	Low
Notification Services	Single Instance	Multiple Business Domains	Low	Medium	Medium
Address Validation	Single Instance	Onboarding, Profile Management	Low	Low	Low
Transaction Validation	Single Instance	Payment, Transfer Services	Medium	Medium	Medium
Fraud Detection	Single Instance	All Transaction Services	Medium	High	High
Currency Conversion	Single Instance	International Transaction Services	Low	Low	Low
Document Generation	Single Instance	Reporting, Customer Communications	Medium	Medium	Medium
Audit Logging	Distributed Instances	All Services	Very Low	Medium	Medium

Table 3: Service Reusability Metrics Across Financial Microservices Architecture [7, 8]

10.48047/jocaaa.2025.34.12.32

5. Cloud Infrastructure and Operational Patterns

5.1 Containerization and Deployment Strategies

Container technology ensures that micro services have a consistent execution environment between the development, testing, and production environments. All of the services are bundled together with runtime dependencies, libraries, and configuration into immutable images that can be deployed with the same effect on any underlying infrastructure. This consistency eradicates any environment-specific problems and accelerates the deployment cycles because it ensures that the tested artifacts are delivered to production in an untouched state. Container orchestration systems handle the containerized service life cycle and automate deployment, scaling, and networking, as well as recovery. Declarative configuration defines the desired state-number of instances, resource needs, health screen measures-and the orchestration platform constantly aligns real state to desired state. The research conducted on the topic of container technology adoption into the microservices architecture earlier revealed that it has several benefits: enhancement of resource utilization, support of deployment processes, greater portability in the context of the infrastructure, and reduction of the burden on operations in comparison with the traditional methods of virtualization [9]. By doing so, enhanced automation permits deployment frameworks, including blue-green releases, canary releases, and rolling updates, that cut down on the downtime and risk in updating services.

To scale elastically, instances of services have to be stateless. Externalizing the state into databases, caches, or distributed storage enables any service instance to handle any request independently. In turn, this allows horizontal scaling free of session affinity concerns. Financial services that have strong consistency needs may leverage distributed coordination services or database transaction capabilities while maintaining stateless application logic. Configuration management via environment variables or configuration services enables a single service image to operate in environments with appropriate settings.

5.2 Service Mesh Integration

The service mesh infrastructure deals with cross-cutting issues of microservices architectures, offering features like traffic management to security enforcement, and observability without service code modification. Sidecar proxies that are deployed with every instance of a service intercept network communication, applying policies and gathering telemetry in a transparent manner to the service. The control plane components manage proxy configuration, propagating updates as policies change. Intelligent load balancing, circuit breaking, enforcing timeouts, and retry logic are all supported traffic management features. The features enhance resilience in the system by ensuring cascading failure is avoided, as well as automatically routing around unhealthy instances. Circuit breaking, in particular, stops requests to failing downstream services, giving them a chance to recover while maintaining system stability, which is especially desirable in financial services. Recently, it was discovered that service mesh platforms address critical challenges in a distributed system, such as service discovery, load balancing, failure recovery, security policy enforcement, and observability, yet also open avenues for research in performance optimizations and standardization. Retry policies have to be carefully designed with respect to idempotency, since a financial transaction cannot occur twice.

Coordination Pattern	Financial Workflow	Services Involved	Response Time Requirement	Consistency Model	Failure Recovery Method
Orchestration	Payment Processing	6 services	Real-time (sub-	Strong	Compensati

10.48047/jocaaa.2025.34.12.32

			second)	Consistency	ng Transaction
	Loan Approval	8 services	Near Real-time (seconds)	Strong Consistency	Saga Pattern
Choreography	Account Balance Update	4 services	Asynchronous	Eventual Consistency	Event Replay
	Fraud Analysis	5 services	Asynchronous	Eventual Consistency	Idempotent Processing
Hybrid	Customer Onboarding	10 services	Mixed (sync + async)	Eventual Consistency	Partial Rollback
	Transaction Reconciliation	7 services	Batch Processing	Eventual Consistency	Manual Intervention
Orchestration	Fund Transfer	5 services	Real-time (sub-second)	Strong Consistency	Two-Phase Rollback
	Notification Delivery	3 services	Asynchronous	Eventual Consistency	Retry with Backoff

Table 4: Workflow Coordination Pattern Performance in Financial Operations [9, 10]

6. Governance and Compliance Considerations

6.1 Regulatory Reporting and Audit Trails

The financial institutions are also under a wide scope of regulation and have a wide scope of reporting and rigorous audit trail requirements. Such requirements should be applied to microservices architectures without compromising the autonomy of services and the performance of the system. The audit requirements are compatible with event-driven architectures, where all significant state transitions occur as domain events, which are append-only logs of the activity taking place in the system. Aggregation of several services is usually needed to complete regulatory reporting, and it is difficult to achieve when using micro services since information is distributed amongst databases. Dedicated reporting services consume domain events from operational services, building specialized read models that optimize for regulatory queries. These reporting services store historical data for the length of required retention periods, possibly using different storage technologies than are used by operational services. Data lineage tracking ensures that numbers being reported can be traced back to source transactions, satisfying the regulatory need for verifiability. Immutability is a core requirement for many financial systems, whereby historical records should not be modified or deleted. Event sourcing naturally ensures immutability since every event is treated as a permanent historical fact. Studies that investigated challenges in implementing event sourcing have found that while the pattern offers substantial advantages for audit trails and temporal queries, artifacts like event schema evolution, strategies to convert data across versions, and long-term maintainability of event stores must be carefully managed to ensure that historical events remain interpretable [11]. All persistence services that are traditional and state-based should have audit logging functionality, which will record all changes in states with time, users, and changed values. Tamper evidence can be provided through cryptographic algorithms, e.g., the hash chain or the digital signature, to verify the integrity of the historical records.

10.48047/jocaaa.2025.34.12.32

6.2 Security Architecture

MLOps expands the DevOps concepts to machine learning by offering reproducible model training and validation, deployment, and monitoring. Microservice-based models reveal prediction interfaces, through which business services can access ML capabilities without necessarily implementing complex algorithms themselves. Principles of zero-trust security assume that network position does not confer trust, and authentication and authorization of all service interactions are required regardless of network location. Identity and access management integrates with institutional identity providers, authenticating users and services through standard protocols such as OAuth 2.0 and OpenID Connect. Token-based authentication allows services to validate requests without frequent calls to centralized authentication services for better performance and resilience. An analysis of security implementations in microservices environments identified key architectural considerations involving distributed identity management, secure protocols for inter-service communications, defense-in-depth strategies to protect sensitive data, and holistic monitoring mechanisms that allow visibility across distributed security boundaries [12]. Short-lived access tokens limit exposure from token theft, and refresh tokens allow long-lived sessions without users being required to authenticate multiple times. Certificate-based authentication is utilized for machine-to-machine communication service accounts, while automated rotation of these certificates prevents the long-term exposure of credentials.

7. Future Directions and Emerging Patterns

7.1 Integration of AI and Machine Learning

Machine learning is moving to financial services, with financial institutions, such as fraud detection, credit scoring, customer service automation, and individualized recommendations, getting it embedded into services. MLOps applications apply the DevOps ethos of machine learning to give model training, validation, deployment, and monitoring routines. The business services access the ML capabilities, but do not write complex algorithms, by invoking models deployed as microservices. Feature stores provide centralized management of machine learning features to ensure consistency between training and inference. Services publish feature data to stores, thus making them available for model training, and also support low-latency retrieval when performing real-time prediction. Feature versioning enables the retraining of models with historical feature definitions; that is, reproduction of their conditions and support for model comparisons. Feature stores help financial institutions standardize how customer behavior, transaction patterns, and market conditions are represented across different models. Model monitoring detects the degradation of prediction quality, triggering either retraining or an alert when models are no longer performing adequately. Financial environments evolve continuously; thus, models that are trained on historical data lose accuracy in a progressive manner. Various research related to deep learning methodologies has provided initial techniques on neural network architecture for advanced pattern recognition ability in complicated datasets, which supply the computational frameworks underlying modern machine learning applications across domains, including financial services 13. Monitoring compares the predictions with the real results and estimates the accuracy measures and statistical drift in the input distributions.

7.2 Evolution of Event-Driven Architecture

Event streaming systems offer guaranteed, sequenced records of domain events, allowing a broad range of consumption behaviors not limited to simple pub-sub messaging. Event sourcing is an obvious extension of streaming platforms where services emit events into streams as archives of events. Other services then consume those streams, building projections, triggering workflows, or maintaining real-time analytics.

10.48047/jocaaa.2025.34.12.32

Financial transaction streams might feed fraud detection systems, account reconciliation processes, and customer notification services, all from the same event logs. Complex event processing processes real-time streams of events by searching for patterns among various events that indicate that a serious situation has transpired. Fraud identification can compare the occurrences in a set of accounts and showcase suspicious trends that are not evident when the few accounts are individually analyzed. Various studies investigating microservices communication patterns have shown that event-driven architectures can offer significant benefits in the case of distributed systems by using asynchronous messaging, decreasing the coupling between services, improving scalability through event-based coordination, and making systems more resilient by avoiding synchronous dependencies [14]. Market surveillance systems monitor trade events, recognizing potentially manipulative patterns that require regulatory reporting.

Conclusion

The integration of Domain-Driven Design with BIAN service domain definitions offers a structured approach toward the architecture of financial microservices that solves both business alignment and technical excellence. Bounded contexts, ubiquitous languages, and tactical patterns ensure that services reflect true business domains rather than arbitrary technical divisions. Normalized descriptions of services accelerate the process of making decisions about decomposition and make them interoperable across institutions and vendors. These frameworks, when combined, have been used to steer the evolution of monolithic architectures towards modular cloud-native architectures that are innovative yet maintain reliability, a key requirement for financial services. The structure of an organization, development practices, and cultural evolution are the inseparable components of the successful implementation, not just the technical architecture. End-to-end ownership of services in cross-functional teams enables the teams to be autonomous, and platform teams offer common capabilities to eliminate unnecessary duplication. To ensure that the stability of the system is sustained, automated testing, continuous integration, and advanced deployment strategies are used to ensure that frequent changes are made to the system. Issues of security and compliance will be felt throughout the implementation process. They are still converging with another emerging technology, such as machine learning, event streaming, and edge computing, transforming the architecture of financial services. The approach of basing architecture choices on business domain knowledge and using industry-standard service definitions can guide institutions to evolve into modern architectures that enhance their flexibility, reliability, resource use, and incorporation of new technologies-meaning direct competitive advantage through accelerated innovation and quality customer experiences.

References

[1] Matthias Galster and Paris Avgeriou, "Empirically-grounded reference architectures: A proposal," QoSA-ISARCS '11: Proceedings of the joint ACM SIGSOFT conference -- QoSA and ACM SIGSOFT symposium -- ISARCS on Quality of software architectures -- QoSA and architecting critical systems -- ISARCS, 2011. [Online]. Available: https://dl.acm.org/doi/10.1145/2000259.2000285

[2] BIAN, "Service Landscape," Banking Industry Architecture Network. [Online]. Available: https://bian.org/deliverables/service-landscape/

[3] Cesare Pautasso et al., "Microservices in Practice, Part 1: Reality Check and Service Design," IEEE Software, Volume 34, Issue 1, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7819415

[4] BIAN, "BIAN MetaModel Overview,". [Online]. Available: https://bian.org/servicelandscape-12-0-0/views/view_52292.html

10.48047/jocaaa.2025.34.12.32

[5] Sara Hassan et al., "Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity," ResearchGate, 2017. [Online]. Available: https://www.researchgate.net/publication/316061076_Microservice_Ambients_An_Architectural_Meta-Modelling_Approach_for_Microservice_Granularity
[6] BIAN, "Behavior Qualifier Type," [Online]. Available: https://bian.org/servicelandscape-9-0/views/view_165298.html
[7] Davide Taibi et al., "Architectural Patterns for Microservices: A Systematic Mapping Study,". [Online]. Available: https://www.scitepress.org/papers/2018/67983/67983.pdf
[8] BIAN, "Service Operation,". [Online]. Available: https://bian.org/servicelandscape-9-0/object_19.html?object=58764
[9]David Jaramillo et al., "Leveraging microservices architecture by using Docker technology," ResearchGate, 2016. [Online]. Available: https://www.researchgate.net/publication/306064413_Leveraging_microservices_architecture_by_using_Docker_technology
[10] Wubin Li et al., "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8705911
[11] Michiel Overeem et al., "The dark side of event sourcing: Managing data conversion," ResearchGate, 2017. [Online]. Available: https://www.researchgate.net/publication/315637858_The_dark_side_of_event_sourcing_Managing_data_conversion
[12] Tetiana Yarygina and Anya Helene Bagge, "Overcoming Security Challenges in Microservice Architectures," 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8359144
[13] Jürgen Schmidhuber, "Deep learning in neural networks: An overview, "Neural Networks, Volume 61, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0893608014002135
[14] Muhammad Waseem et al., "A Systematic Mapping Study on Microservices Architecture in DevOps, "Journal of Systems and Software, Volume 170, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0164121220302053