International Journal of Multidisciplinary Research and Growth Evaluation.

# Developing a Conceptual Model for Cross-Domain Microservices Using Event-Driven and Domain-Driven Design

**Teemu Myllynen [1*], Eunice Kamau [2], Sikirat Damilola Mustapha [3], Gideon Opeyemi Babatunde [4], Abidemi Adeleye Alabi [5]**

[1] Independent Researcher, Helsinki, Finland
[2] Independent Researcher, Dallas, Texas, USA
[3] Montclair State University, Montclair, New Jersey, USA
[4] Cadillac Fairview, Ontario, Canada
[5] Ericsson Telecommunications Inc., Lagos, Nigeria

* Corresponding Author: **Teemu Myllynen**

## Article Info

**ISSN (online):** 2582-7138
**Volume:** 04
**Issue:** 01
**January-February 2023**
**Received:** 11-12-2022
**Accepted:** 04-01-2023
**Page No:** 635-638

**Abstract**
The growing complexity of modern software systems necessitates architectural approaches that enhance scalability, flexibility, and maintainability. Microservices architecture has emerged as a leading solution, particularly for cross-domain applications. This paper presents a conceptual model for cross-domain microservices by integrating Event-Driven Design (EDD) and Domain-Driven Design (DDD) principles. The proposed model focuses on structuring microservices around distinct bounded contexts, enabling modular development and fostering a deep alignment between technical architecture and business objectives. By employing EDD, the model leverages asynchronous communication and event streaming to decouple microservices, ensuring resilience and real-time responsiveness across domains. DDD is utilized to define domain boundaries, prioritize business logic, and establish ubiquitous language, facilitating seamless collaboration between technical and non-technical stakeholders. The model also introduces a unified framework for orchestrating complex interactions across microservices, enhancing system coherence without compromising autonomy. Key components of the conceptual model include event sourcing for maintaining robust data consistency, CQRS (Command Query Responsibility Segregation) to separate read and write operations, and aggregate design to manage domain entities efficiently. The model emphasizes event choreography over centralized orchestration, promoting loose coupling and scalability. Integration challenges, such as data synchronization, eventual consistency, and cross-domain event propagation, are addressed with practical guidelines for implementing resilient patterns. A comparative analysis with traditional service-oriented architectures underscores the model's advantages in supporting diverse business needs and rapid adaptability to changing requirements. The proposed conceptual model is validated through case studies in e-commerce and financial services, demonstrating its potential to optimize operational workflows and enhance user experiences. This research contributes to the growing discourse on microservices by bridging the gap between technical and business domains, offering a robust framework for designing cross-domain systems. Future work will explore extending the model with advanced AI-driven decision-making and monitoring capabilities.

**DOI:** https://doi.org/10.54660/.IJMRGE.2023.4.1.635-638

**Keywords:** Microservices Architecture, Event-Driven Design, Domain-Driven Design, Cross-Domain Systems, Bounded Context, CQRS, Event Sourcing, Scalability, Resilience, Asynchronous Communication

## 1. Introduction

Cross-domain software systems face significant challenges due to the complexity of managing diverse business functions, systems, and data flows. These challenges become even more pronounced when dealing with scalability, flexibility, and maintainability, all of which are essential for ensuring that the system can evolve without sacrificing performance or reliability.

Traditional monolithic architectures often struggle to address these issues, leading to longer development cycles, difficulties in adapting to new business requirements, and increased maintenance overhead (Ahmaro, Abualkishik & Yusoff, 2014, Malik, 2015). Microservices architecture offers a promising solution, but when applied to cross-domain scenarios, it requires careful consideration of how different domains interact, communicate, and integrate without compromising autonomy.

Event-Driven Design (EDD) and Domain-Driven Design (DDD) provide complementary frameworks that address these challenges. EDD promotes decoupling services through asynchronous communication and event propagation, ensuring that different services can operate independently while reacting to changes in real-time. DDD, on the other hand, focuses on aligning software architecture with business goals by defining clear bounded contexts and developing a shared understanding among stakeholders (Ali, *et al*., 2018, Martinetti, Schakel & van Dongen, 2018). The integration of these two approaches can lead to a more robust architecture for cross-domain microservices, enabling seamless communication and modularization while maintaining system coherence.

The objective of this work is to propose a conceptual model that integrates both EDD and DDD to optimize the design of cross-domain microservices. The model aims to improve the efficiency of cross-domain communication, enhance system modularity, and ensure that business requirements are met without sacrificing flexibility or performance. By incorporating event-driven architectures, the model can ensure real-time responsiveness and scalability across microservices, while DDD will provide the necessary framework for aligning technical solutions with business needs (Andriyanto & Doss, 2021, Morrell, *et al*., 2021).

This study focuses on business-critical applications, particularly in sectors such as e-commerce and financial services, where maintaining system reliability, scalability, and flexibility is essential for competitiveness. These domains often require complex, real-time interactions between multiple services, making the integration of EDD and DDD particularly beneficial. Through the proposed conceptual model, this research aims to bridge the gap between technical architecture and business objectives, offering a framework for the development of scalable, maintainable, and flexible cross-domain microservices systems.

### 2.1. Literature Review

Microservices architecture has emerged as a transformative approach to building software systems, providing a solution to the limitations of monolithic structures. In a microservices architecture, a system is divided into a set of loosely coupled, independently deployable services, each responsible for a specific business function or domain. This modularity allows teams to develop, deploy, and scale each service independently, reducing the complexity of managing large and monolithic applications (Bangemann, *et al*., 2014, Mustalahti & Rakotonarivo, 2014). The core principles of microservices include autonomy, scalability, flexibility, and resilience. Each microservice typically focuses on a specific business capability, and the interactions between services are designed to be as decoupled as possible, often through lightweight communication mechanisms such as RESTful APIs, messaging queues, or event streams.

One of the major advantages of microservices over traditional monolithic systems is the ability to scale services independently. This means that when certain components of a system experience increased load, they can be scaled without affecting the entire application. Moreover, microservices can be developed and deployed by independent teams, which accelerates development cycles and improves responsiveness to changing business needs. Microservices also enable the use of different technologies for different services, allowing developers to select the best tools for each specific problem. Additionally, since each service operates independently, failures in one service do not necessarily impact others, which increases system resilience. Figure 1 shows Functional elements of the microservices based IoT architecture as presented by Datta & Bonnet, 2018).

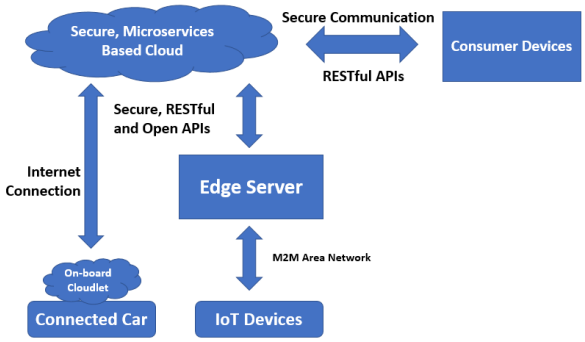


**Fig 1:** Functional elements of the microservices based IoT architecture (Datta & Bonnet, 2018).

Despite the many benefits, building cross-domain microservices systems introduces significant challenges. As systems grow in complexity, managing interactions between services across different business domains becomes difficult. While microservices are designed to be independent, cross-domain communication is often required, and this is where event-driven and domain-driven designs come into play (Baxter, 2016, N'guessan, Achiepo & Diako, 2023). Event-Driven Design (EDD) focuses on the use of events to decouple services and facilitate communication. EDD relies on asynchronous communication, where services react to events triggered by other components, rather than waiting for direct requests. This approach improves the flexibility and scalability of systems, particularly in environments where rapid changes are required.

In EDD, events act as messages that indicate a change of state in a system. These events can be used to trigger actions across different microservices, ensuring that each service responds to the change in real-time without needing to be tightly coupled to other services. One of the key benefits of EDD is that it allows for high levels of decoupling between services. Each service can emit events when it undergoes a state change, and other services can listen to these events to perform necessary actions, all without having direct dependencies on each other. This is particularly beneficial in cross-domain systems where services from different domains need to communicate without knowing the internal workings of each other.

Event sourcing is a powerful pattern that often accompanies EDD. In event sourcing, changes to the state of an application are captured as a series of immutable events, rather than directly modifying the system's state. This approach provides a complete audit trail of all changes, making it possible to

replay events to reconstruct the system's state at any point in time. Event sourcing is often used in combination with Command Query Responsibility Segregation (CQRS), which separates the processes of reading and writing data (Bellemare, 2020, Nalla & Reddy, 2021). This separation allows for optimized handling of both commands (actions that modify data) and queries (actions that retrieve data), improving system performance and scalability. Together, event sourcing and CQRS enable systems to handle complex workflows with high levels of performance and consistency.

Alongside EDD, Domain-Driven Design (DDD) provides a structured approach to building software systems that reflect the complexities of the business domain. DDD encourages developers to focus on the core business concepts and to use a shared vocabulary, referred to as ubiquitous language, to ensure alignment between business experts and developers. By developing a model that reflects the business's needs, DDD helps ensure that the software system accurately represents the domain, making it easier to evolve as business requirements change. Datta & Bonnet, 2018, presented Micro elements in the IoT architecture as shown in figure 2.

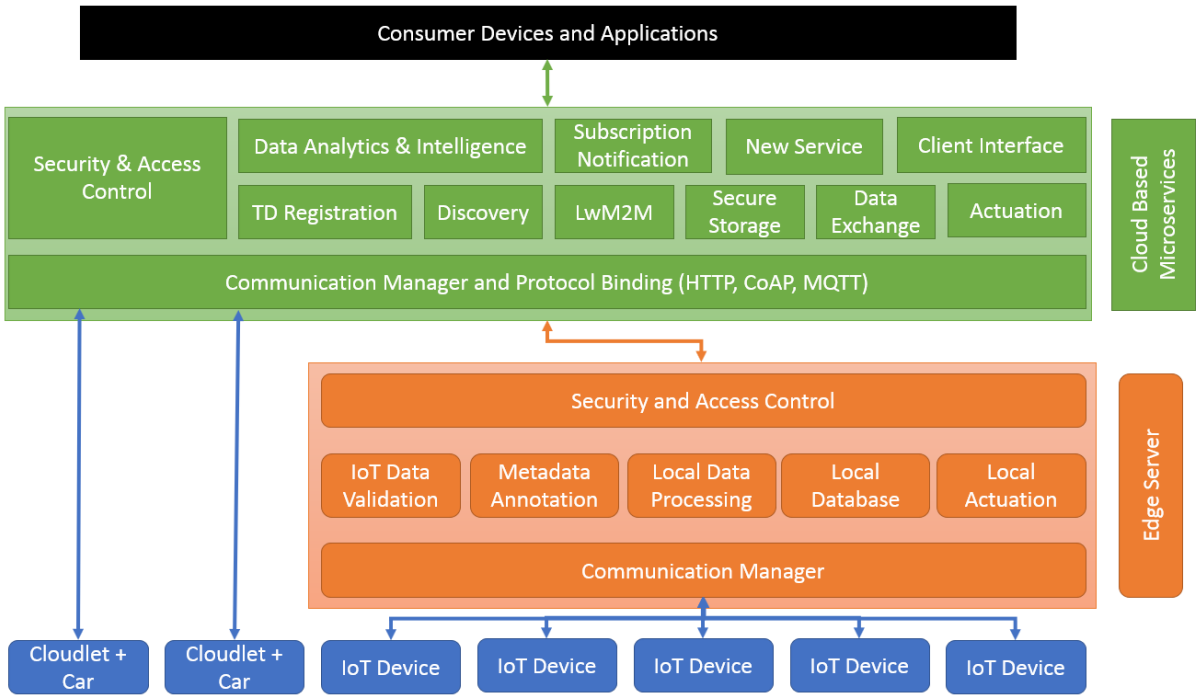


**Fig 2:** Micro elements in the IoT architecture (Datta & Bonnet, 2018).

One of the key concepts in DDD is bounded contexts. A bounded context defines the boundaries within which a particular domain model applies. In a microservices context, each microservice typically corresponds to a bounded context, and the boundaries of these services align with specific business capabilities. By clearly defining the boundaries of each service, DDD enables teams to work independently on different services, without fear of overlap or confusion. This is particularly important in cross-domain systems, where multiple microservices from different domains need to interact. Bounded contexts ensure that each service's internal model is well-defined, and the interaction between different services can be managed through well-understood interfaces, such as events or APIs.

Another essential concept in DDD is aggregate design. An aggregate is a cluster of domain objects that are treated as a single unit. Aggregates ensure consistency within a bounded context by enforcing business rules and invariants. In the context of microservices, aggregates play an important role in maintaining consistency within each service. Each service is responsible for managing its own aggregate, and this responsibility helps to minimize the need for direct coordination between services, reducing the risk of conflicts and inconsistencies.

Domain events are another key component of DDD. A domain event represents a significant change in the state of the system that is important to the business. These events can be used to communicate changes across bounded contexts, and when combined with EDD, they allow for seamless communication between microservices (Cerqueus & Delorme, 2023, Newman, 2019). In cross-domain systems, domain events help maintain consistency and synchronization between services, even as they operate independently. They ensure that each service reacts to business-relevant changes in a timely and decoupled manner, supporting the overall cohesion of the system.

Building cross-domain systems comes with inherent challenges, particularly in terms of synchronization, consistency, and scalability. Cross-domain communication often requires the integration of services that may have different models, databases, or workflows. Ensuring data consistency across these services can be difficult, particularly in systems that rely on eventual consistency, as microservices are often designed to be loosely coupled and independently scalable. Techniques like event sourcing and CQRS help address these challenges by ensuring that all changes to system state are captured as events and that each service has its own model of the data it is responsible for, reducing the need for direct synchronization.

Scalability also becomes a significant concern in cross-domain systems. As the number of services and events increases, the ability to scale each service independently becomes critical. Event-driven architectures naturally lend themselves to scalability, as events can be processed in

parallel by different services, allowing the system to handle increased load. However, ensuring that events are processed in the correct order and that they are not lost during communication is another challenge that must be addressed (Classi, *et al.*, 2018, Nguyen, 2019). Technologies like message queues, event brokers, and stream processing platforms play a crucial role in ensuring the reliable delivery and processing of events.

Finally, managing the overall coordination and orchestration of services in a cross-domain system is complex. Event-driven approaches help by reducing the need for centralized control, but they also introduce new challenges related to maintaining coherence and managing the flow of events. Achieving this balance between autonomy and coordination is one of the key challenges in designing cross-domain microservices systems, and integrating DDD principles with EDD can provide a robust framework for managing these complexities. By focusing on clear service boundaries, domain events, and asynchronous communication, developers can create systems that are both scalable and maintainable, with the flexibility to adapt to changing business needs.

### 2.2. Proposed Conceptual Model

The proposed conceptual model for cross-domain microservices integrates Event-Driven Design (EDD) and Domain-Driven Design (DDD) to address the complexities of building scalable, maintainable, and flexible systems. The model aims to provide a structured approach for developing systems that can evolve in response to changing business requirements while ensuring efficient communication and data consistency across services. By combining these two approaches, the model facilitates autonomous, loosely coupled services that communicate effectively within and across domains.

The core components of this conceptual model include bounded contexts, aggregates, events, and services. Each of these components plays a crucial role in ensuring that the system adheres to both the business and technical requirements. Bounded contexts, as defined in DDD, represent the boundaries within which a specific domain model applies. These contexts are key to organizing microservices and ensuring that each service operates independently (Bello, *et al.*, 2023, Dalibor, *et al.*, 2022, Nimmagadda, 2023). Aggregates, another critical element of DDD, represent the core business entities within a bounded context. In the model, aggregates ensure that business rules and data consistency are maintained within each service, and they form the foundation upon which events are generated and propagated.

The integration of EDD and DDD is central to the proposed model. EDD emphasizes the decoupling of services through event-based communication, ensuring that services can react to changes in the system asynchronously. In combination with DDD's focus on aligning the system with business objectives, the model ensures that each service not only remains independent but is also tightly aligned with the business domain it serves. The interactions between services occur via events that are published when an aggregate undergoes a state change, and other services subscribe to these events to react accordingly. This design supports system scalability and resilience by enabling each service to independently evolve without disrupting the overall system.

The architectural layers of the proposed model are divided into three key areas: the domain layer, the application layer, and the infrastructure layer. The domain layer is where the core business logic resides, encompassing the bounded contexts, aggregates, and domain events. Each bounded context is responsible for its own set of aggregates, which enforce the business rules specific to that context. The aggregates manage the internal state of the domain and trigger domain events when changes occur (Debski, *et al.*, 2017, Özkan, Babur & Brand, 2023). These domain events represent significant occurrences within the system, such as the completion of an order or the status update of a transaction. Domain events are crucial for maintaining data consistency across services, as they propagate changes asynchronously across the system.

The application layer focuses on orchestrating the flow of events between services. In this layer, event choreography and orchestration are employed to define how events are handled. Choreography involves each service autonomously subscribing to and publishing events, while orchestration involves a central service coordinating the flow of events between services. Both approaches have their merits, with choreography promoting loose coupling between services and orchestration providing more control over the flow of events. The application layer, therefore, ensures that events are processed in the correct order and that services react appropriately to state changes in other services.

The infrastructure layer supports the communication and data propagation between services. This layer includes the event bus and message brokers that handle the delivery and processing of events. The event bus acts as a conduit through which events flow from one service to another, while message brokers ensure reliable delivery, message persistence, and event ordering. These components enable asynchronous communication and ensure that services can operate independently while staying synchronized through the propagation of events.

Communication patterns such as event sourcing and Command Query Responsibility Segregation (CQRS) are fundamental to ensuring consistency and scalability in the model. Event sourcing is a technique where all state changes in the system are captured as events, which are stored in an event store. By maintaining an immutable history of events, event sourcing provides a complete audit trail of system changes, enabling services to reconstruct their state at any point in time (Diatte, *et al.*, 2022, Prasat, *et al.*,2022). This is particularly important for ensuring consistency in cross-domain microservices systems, where different services may have different views of the same data. Event sourcing allows services to maintain their own local state while ensuring that they remain consistent with the events occurring across the system.

CQRS is another key pattern used in the proposed model to separate the concerns of reading and writing data. In a traditional system, the same data model is used for both read and write operations. However, in a microservices context, separating these concerns allows for more efficient and scalable handling of data. In the proposed model, write operations are handled through commands, which trigger state changes and generate events, while read operations are optimized through queries that retrieve data from read-optimized views of the system. This separation not only improves performance but also ensures that the system can handle different levels of load for reading and writing operations. Overall representation of the microservices

workflow and data flow as presented by Labiadh, *et al*., 2021, is shown in figure 3.

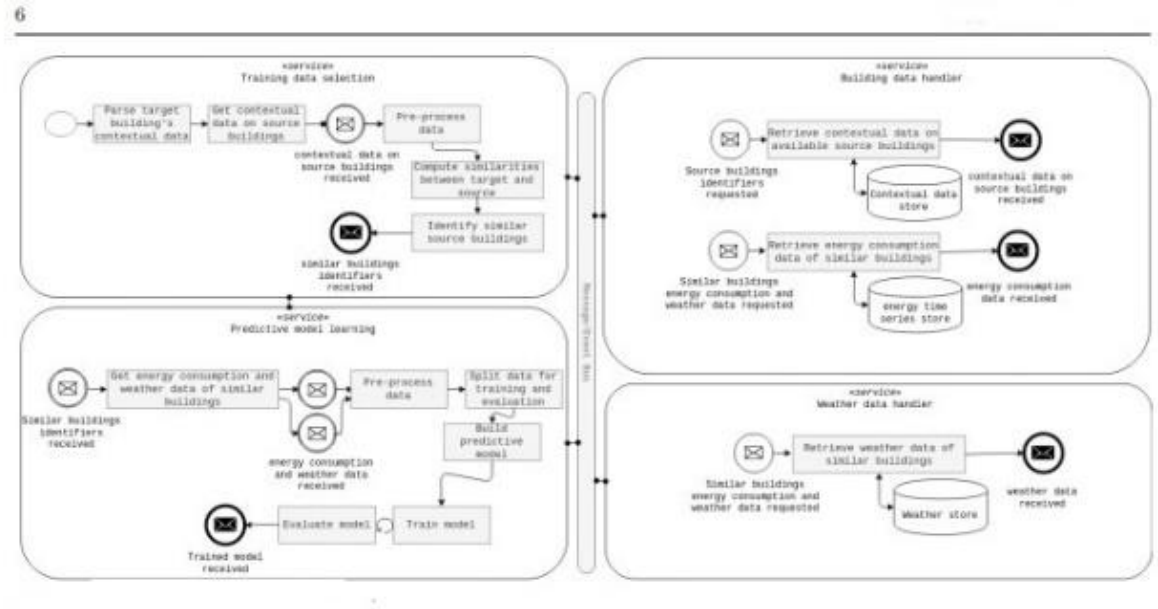


**Fig 3:** Overall representation of the microservices workflow and data flow (Labiadh, *et al*., 2021).

To ensure the effective propagation of events across domains, the model employs strategies for cross-domain event propagation, including choreography and orchestration. Choreography allows services to independently subscribe to and react to events, promoting flexibility and autonomy. Each service in this model can emit events when a state change occurs and other services can listen to those events to trigger necessary actions (Dobaj, *et al*., 2018, Pyykkö, Suoheimo & Walter, 2021). This creates a highly decoupled system where services do not depend on each other's internal state, but instead react to the changes conveyed by events. Choreography is particularly useful when the system needs to be highly scalable, as it allows for distributed processing of events.

On the other hand, orchestration involves a central service that coordinates the flow of events between microservices. This approach provides more control over the event flow, ensuring that events are processed in a predefined order. Orchestration can be beneficial in situations where a specific sequence of actions needs to be performed in response to events, or when a high level of coordination between services is required. The choice between choreography and orchestration depends on the specific requirements of the system and the degree of independence and coordination needed between services (Sturtevant, *et al*., 2022, Vallejo-Vaz, *et al*., 2016).

Event versioning and schema evolution are also critical considerations in the context of cross-domain microservices. As the system evolves, the structure of events may change, and ensuring backward compatibility is crucial to maintaining the system's stability. Event versioning involves managing different versions of events over time, ensuring that new versions of events can coexist with older ones. This enables services to continue operating without disruption, even as new functionality is introduced. Schema evolution is closely related to event versioning, as changes to the structure of events require careful management to avoid breaking existing services. Tools such as schema registries can be used to manage event schemas and ensure that all services are aware of the changes in the event structure.

The proposed conceptual model provides a comprehensive framework for building cross-domain microservices systems that are scalable, flexible, and maintainable. By integrating EDD and DDD, the model ensures that services are decoupled and aligned with business needs while maintaining data consistency and scalability. The architectural layers of domain, application, and infrastructure, along with communication patterns such as event sourcing and CQRS, provide a robust structure for developing and managing cross-domain systems (Ehikioya & Guillemot, 2020, Rafique, Khan & Dou, 2019). Strategies for cross-domain event propagation, including choreography and orchestration, offer the flexibility to choose the most appropriate communication pattern for the system's needs, while event versioning and schema evolution ensure that the system can evolve without breaking. This conceptual model is a valuable approach for developing complex systems that need to handle real-time data across multiple domains while remaining agile and responsive to business changes.

### 2.3. Methodology

The methodology for developing a conceptual model for cross-domain microservices using Event-Driven Design (EDD) and Domain-Driven Design (DDD) follows a systematic and structured approach to address the complexities of scalable and maintainable system architectures. The research approach is rooted in design science methodology, which is a well-established method for creating, analyzing, and validating design artifacts such as conceptual models. This methodology allows for the development of practical solutions to real-world problems while contributing to the advancement of knowledge in the field. In the case of cross-domain microservices, the design science approach provides a clear framework for conceptualizing and testing the integration of EDD and DDD principles in a way that addresses the challenges of cross-domain communication, scalability, and system evolution.

The first step in the methodology is a comprehensive review of existing architectures and frameworks that are relevant to microservices, EDD, and DDD. This comparative analysis

helps identify gaps and limitations in current approaches, offering a foundation for the development of a novel conceptual model. By examining existing systems in various industries, such as e-commerce and financial services, the research identifies the challenges these systems face, including difficulties in ensuring consistency, maintaining data integrity, and enabling effective communication between services spread across different domains (Fedushko & Ustyianovych, 2022, Razzaq & Ghayyur, 2023). This comparative study not only provides insights into the strengths and weaknesses of existing architectures but also informs the design of the proposed conceptual model by revealing the specific needs of cross-domain systems.

The conceptualization of the model is based on the core principles of EDD and DDD. EDD, with its emphasis on asynchronous communication and event-based interactions between services, is crucial for ensuring that the services remain decoupled and responsive to changes in state. DDD, on the other hand, provides a structured approach to organizing business logic through bounded contexts, aggregates, and domain events (Labiadh, *et al*., 2021). By combining these two approaches, the model integrates event-driven communication patterns with domain-specific business logic to create a system that is both flexible and scalable. The conceptual model is designed to be modular, with each domain having its own bounded context and aggregate structure, and events serving as the primary means of communication between domains. This structure ensures that each service operates autonomously while still maintaining a high level of coordination across the system (Navarro, 2017, Pestana, 2023).

Once the conceptual model is developed, the next step is to design and implement prototypes based on the model in real-world domains, such as e-commerce and financial systems. These domains are chosen because of their complex and dynamic nature, which makes them ideal candidates for testing the scalability and flexibility of the proposed model. The prototypes serve as proof of concept, demonstrating how the model can be applied to solve the challenges faced by cross-domain systems (Elujide, *et al*., 2021, Gade, 2021, Ribeiro, 2022). These prototypes are designed to implement key features of the model, such as event sourcing for maintaining state consistency, CQRS for optimizing read and write operations, and the integration of event-driven communication between services. The implementation phase also involves testing the model's performance, ensuring that the architecture can scale to handle large volumes of data and user interactions while maintaining reliability and responsiveness.

The model is then validated using a combination of performance and functionality tests. Scalability, maintainability, and domain alignment are the primary criteria for evaluation. Scalability is assessed by simulating various load conditions to determine how the system performs under different levels of demand. The goal is to ensure that the system can handle growth in terms of both data volume and the number of services without compromising performance (Laranjeiro, Soydemir & Bernardino, 2015). Maintainability is evaluated by examining how easily the system can be modified and extended to accommodate new business requirements. This is crucial in cross-domain systems, where business logic may evolve over time, and the system must remain flexible to accommodate these changes (Guzmán & González de Molina, 2015, Rishaug & Wika, 2018). Domain alignment is evaluated by assessing how well the model reflects the business requirements of each domain. This is ensured by verifying that the bounded contexts and aggregates accurately represent the real-world concepts of the business domain and that the communication between services aligns with the domain's needs.

To measure the performance and communication efficiency of the proposed model, a set of key performance indicators (KPIs) is defined. These KPIs include response time, throughput, and resource utilization, which provide a clear picture of how well the system performs under different conditions. Response time measures the time it takes for a service to process an event and send a response, while throughput measures the volume of events processed within a given time frame (Özkan, Babur & Brand, 2023). Resource utilization measures the efficiency with which the system uses computational resources, such as CPU, memory, and network bandwidth. These metrics are essential for assessing the efficiency of the system and ensuring that it meets the performance requirements for business-critical applications.

The methodology also involves continuous feedback loops throughout the development and implementation phases. After the prototypes are implemented, feedback is gathered from stakeholders, including system architects, domain experts, and end-users. This feedback is used to refine the conceptual model and make necessary adjustments to improve its functionality and performance (Hashem, *et al*., 2015, Siddiqa, *et al*., 2016). By involving stakeholders early in the process, the methodology ensures that the final model aligns with the practical needs of the business and the technical requirements of the system.

In addition to the performance-based evaluation, the methodology includes a detailed analysis of the model's ability to handle cross-domain communication. One of the core challenges of cross-domain systems is ensuring that services can communicate effectively while maintaining their independence. The proposed model leverages EDD and DDD to address this challenge by ensuring that events serve as the primary means of communication, allowing services to remain decoupled while still responding to changes in other domains (Hahn & Lee, 2020, Sabou, *et al*., 2020). The evaluation of cross-domain communication focuses on ensuring that the event propagation mechanisms are efficient and reliable, with events being delivered in a timely manner to all relevant services. This is achieved by using event buses and message brokers, which provide a reliable infrastructure for event delivery.

Finally, the methodology involves considering the long-term sustainability of the model. As the system evolves and new services are added, it is crucial that the model remains adaptable to future changes. The scalability and maintainability criteria are particularly important in this regard, as they ensure that the system can continue to grow and evolve without requiring a complete redesign (Saidi, Tissaoui & Faiz, 2023). The methodology also emphasizes the importance of event versioning and schema evolution, which allows the system to adapt to changes in the structure of events without breaking existing services. By incorporating these principles into the design and evaluation phases, the methodology ensures that the proposed conceptual model is not only effective but also sustainable in the long term.

In conclusion, the methodology for developing a conceptual

model for cross-domain microservices using EDD and DDD combines a systematic design approach with rigorous evaluation criteria to ensure the model's scalability, maintainability, and domain alignment. By following this methodology, the research aims to provide a robust solution to the challenges faced by cross-domain systems and offer valuable insights into the design and implementation of scalable, flexible, and maintainable microservices architectures (Hashem, *et al*., 2015, Siddiqa, *et al*., 2016).

### 2.4. Case Studies

Developing a conceptual model for cross-domain microservices using event-driven and domain-driven design (DDD) has gained significant attention due to its capacity to handle complex, distributed systems effectively. Case studies from domains such as e-commerce and financial services illustrate how these principles can address real-world challenges. These domains have unique requirements that emphasize scalability, reliability, and consistency, making them suitable for exploring the application of these architectural paradigms (Elouataoui, *et al*., 2022, Saiod, Van Greunen & Veldsman, 2017).

In the e-commerce domain, managing inventory, orders, and user domains poses inherent complexities due to the interconnected nature of these components. For example, the inventory system must remain synchronized with orders to prevent overselling products. Users interacting with the platform expect a seamless experience, even during high-traffic events such as flash sales or promotional periods (Hassan, Gregory & Li, 2023, Shrestha, Krishna & von Krogh, 2021). By adopting domain-driven design, the architecture decomposes the system into bounded contexts corresponding to inventory, orders, and user management. Each bounded context encapsulates its own domain logic, data, and responsibilities, reducing dependencies and enhancing modularity.

Event-driven communication plays a pivotal role in enabling interactions between these bounded contexts. When a customer places an order, an event is generated and published to an event bus, notifying other services. The inventory service, upon receiving this event, updates stock levels and ensures that inventory availability is accurately reflected (Dal Maso, 2019, Peng, *et al*., 2015). This decoupled communication pattern minimizes direct dependencies between services and facilitates scalability, as services can independently handle their workloads.

Consideration of eventual consistency is critical in such architectures. Unlike traditional monolithic systems where database transactions enforce strict consistency, microservices architectures often embrace eventual consistency to optimize performance and responsiveness. For example, while the order service may immediately confirm an order to the user, the inventory and payment services asynchronously process related updates. This approach requires robust mechanisms to handle failure scenarios, such as retries, compensating transactions, and monitoring to maintain data integrity (Elujide, *et al*., 2021, Intizar, *et al*., 2017, Solberg, 2022).

The adoption of these practices also improves system resilience. E-commerce platforms frequently encounter sudden traffic surges, such as during seasonal sales or product launches. Event-driven architectures allow services to scale independently, handling increased workloads without overloading the entire system. For instance, the order service can scale horizontally to accommodate a higher number of purchase requests, while the inventory service scales based on the complexity of stock updates (Mishra, Komandla & Bandi, 2021).

Turning to the financial services domain, real-time transaction processing presents a unique set of challenges. Financial systems must ensure accuracy, security, and reliability when handling sensitive data such as account balances and payment information. Domain-driven design proves invaluable in this context by isolating concerns into distinct bounded contexts, such as accounts and payments. Each context operates independently while adhering to strict domain-specific rules (Jones, 2014, Kayabay, *et al*., 2022).

In a typical scenario, when a customer initiates a payment, the payment service processes the request and publishes an event indicating the transaction's status. This event notifies the account service, which updates the user's account balance accordingly. By decoupling these operations, the architecture remains flexible and adaptable to evolving requirements. For instance, integrating new payment methods or extending the account system to support loyalty programs becomes less cumbersome due to the modular design (Bello, *et al*., 2023, Joseph, 2023, Strengholt, 2023).

Ensuring consistency between the account and payment domains is paramount in financial services. While eventual consistency is acceptable for certain e-commerce operations, financial systems often demand stronger consistency guarantees (Gökalp, *et al*., 2021, Pora, *et al*., 2020). Techniques such as distributed transactions, sagas, or outbox patterns help coordinate changes across services while avoiding data inconsistencies. For example, the saga pattern orchestrates a sequence of local transactions across services, ensuring that all steps are completed successfully or reverted if any step fails (Datta & Bonnet, 2018). This approach strikes a balance between maintaining consistency and supporting distributed operations.

Event-driven communication further enhances the system's ability to process transactions in real-time. Events such as "Payment Initiated," "Payment Completed," or "Insufficient Funds" provide immediate feedback to the account service, enabling prompt updates to account balances. This real-time responsiveness is crucial for customer satisfaction and operational efficiency, especially in high-frequency trading environments or peer-to-peer payment platforms.

The advantages of event-driven and domain-driven design extend beyond technical considerations to organizational benefits. By aligning the architecture with domain-specific boundaries, development teams can specialize in particular areas, such as accounts or payments, fostering a deep understanding of business rules and processes. This alignment enhances collaboration between technical teams and domain experts, facilitating iterative improvements and innovation (Baxter, 2016).

However, implementing such architectures is not without challenges. The complexity of distributed systems necessitates robust monitoring, observability, and fault-tolerance mechanisms. In financial services, for example, ensuring the reliability of event streams is critical, as missed or duplicated events can lead to significant discrepancies in account balances. Employing distributed tracing, logging, and alerting tools helps identify and resolve issues promptly, ensuring the system's reliability (Bello, *et al*., 2023, Kalske, 2017, Tapia, *et al*., 2020).

The case studies of e-commerce and financial services

demonstrate the transformative potential of combining event-driven and domain-driven design in cross-domain microservices architectures. While the e-commerce domain benefits from enhanced scalability, modularity, and responsiveness, financial services gain consistency, reliability, and adaptability. Both domains highlight the importance of aligning technical architectures with business needs, emphasizing the role of events and bounded contexts in achieving these goals (Curuksu, 2018, Zolnowski, Christiansen & Gudat, 2016). These principles offer a robust foundation for designing systems capable of thriving in complex, dynamic environments.

### 2.5. Discussion

Developing a conceptual model for cross-domain microservices using event-driven and domain-driven design (DDD) presents a transformative approach to building scalable, resilient, and adaptive systems. This architectural strategy integrates the principles of modularity, decoupled communication, and strong domain alignment to address the complexities of modern distributed systems. The proposed model offers significant benefits, but it is not without its challenges. Understanding both its advantages and limitations is essential for its successful implementation and optimization (Becker, *et al*., 2016, Pora, *et al*., 2018).

One of the primary benefits of this model is enhanced modularity, which lies at the heart of microservices and domain-driven design. By dividing the system into bounded contexts that align with specific business domains, each microservice encapsulates its unique logic and responsibilities. This approach reduces interdependencies and simplifies maintenance, enabling developers to modify or update individual services without risking widespread system disruptions (Kempa, *et al*., 2020, Tchernykh, *et al*., 2019). For example, in an e-commerce application, the inventory, order management, and user services operate as independent units, communicating through well-defined event-based mechanisms. This modularity facilitates rapid adaptation to evolving business needs, such as introducing a new product line or modifying pricing algorithms, without requiring a complete overhaul of the system.

Another critical advantage is improved system resilience. Event-driven communication decouples services, allowing them to operate independently. When one service experiences a failure, it does not necessarily propagate across the system, preserving overall functionality. For instance, if the payment processing service in a financial system encounters an error, other services, such as account management or notifications, can continue functioning (Mishra, Komandla & Bandi, 2021). This resilience is further enhanced by the system's ability to scale individual services based on demand. During high-traffic periods, such as promotional sales in e-commerce or tax season in financial services, specific services can scale horizontally to handle increased loads without affecting other parts of the system.

The alignment between business and IT is another significant benefit of the proposed model. Domain-driven design emphasizes collaboration between technical teams and domain experts, ensuring that the architecture accurately reflects business requirements. Bounded contexts map directly to business processes, creating a shared understanding and vocabulary among stakeholders (Khan, 2017, Tidjon, Frappier & Mammar, 2019). This alignment improves communication, reduces misunderstandings, and accelerates the development of features that directly address user needs. For instance, in a banking system, the separation of account and payment services ensures that each domain's unique rules and workflows are respected while fostering better collaboration between software engineers and financial analysts.

However, the implementation of this model comes with notable challenges, chief among them being the complexity of designing and managing a distributed system. Unlike monolithic architectures, where all components reside within a single application, microservices require careful orchestration to ensure seamless interaction among services. Designing bounded contexts that accurately encapsulate domain logic while minimizing interdependencies demands deep domain knowledge and meticulous planning (Dulam, Katari & Allam, 2020). Additionally, the event-driven communication model requires robust infrastructure to handle message routing, delivery guarantees, and failure recovery. Tools like Kafka, RabbitMQ, or AWS SNS/SQS facilitate event streaming but introduce their own set of operational complexities.

Another significant challenge is handling eventual consistency, a fundamental trade-off in distributed systems. Unlike traditional architectures that rely on strict transactional consistency, event-driven microservices embrace eventual consistency to enhance performance and scalability. While this approach reduces latency and supports asynchronous communication, it introduces potential discrepancies between services at any given moment (Korkmaz & Nilsson, 2014, Vernon & Jaskula, 2021). For example, in an e-commerce platform, the inventory service might temporarily reflect incorrect stock levels after an order is placed until the relevant updates propagate through the system. This limitation requires the implementation of compensating mechanisms, such as retries, dead-letter queues, or sagas, to ensure data consistency and integrity.

The complexity of maintaining eventual consistency can also impact user experience. Customers might encounter scenarios where they place an order only to receive a notification later that the item is out of stock due to delays in inventory updates. Addressing such issues involves balancing the need for real-time responsiveness with the constraints of asynchronous processing (Ahmad, *et al*., 2022, Maja & Letaba, 2022). Strategies like implementing optimistic updates, real-time notifications for pending actions, or providing clear communication about delays help mitigate the impact of eventual consistency on end-users.

Operational challenges further complicate the model's implementation. Monitoring and debugging distributed systems are inherently more difficult than with monolithic architectures. Tracing the flow of events across multiple services requires comprehensive observability solutions, such as distributed tracing, log aggregation, and real-time metrics dashboards. These tools provide visibility into system performance and help identify bottlenecks or failures but demand additional effort and expertise to implement effectively (Strengholt, 2023). Moreover, managing the lifecycle of services, including deployment, versioning, and scaling, necessitates a robust DevOps pipeline with automation tools like Kubernetes, Docker, and CI/CD workflows.

Security and governance also pose challenges in cross-domain microservices. The decoupled nature of the architecture increases the attack surface, necessitating strong

authentication, authorization, and encryption mechanisms for inter-service communication. Implementing these safeguards adds complexity and may introduce performance overhead. Similarly, ensuring compliance with regulatory requirements, such as data protection laws, becomes more intricate when data flows across multiple services and bounded contexts (Chinamanagonda, 2022, Pulwarty & Sivakumar, 2014).

Despite these challenges, the potential of the proposed model to revolutionize system design and operation cannot be overstated. Organizations adopting event-driven and domain-driven design gain a strategic advantage by building systems that are not only more agile and resilient but also better aligned with their business goals. The modularity of the architecture allows for incremental development and continuous deployment, enabling faster delivery of features and quicker responses to market demands (Macero, Macero & Anglin, 2017, Weiler & Lomotey, 2022). By addressing the limitations through thoughtful design and leveraging advanced tooling, businesses can unlock the full potential of this approach.

The proposed model also encourages innovation by empowering teams to experiment and iterate on individual services without risking the stability of the entire system. For example, a team managing the payment service in a financial application could integrate a new fraud detection algorithm or machine learning model without affecting other services like account management or transaction history. This flexibility fosters a culture of continuous improvement and experimentation, driving innovation across the organization (Bhaskaran, 2020, Yu, *et al*., 2019).

In conclusion, developing a conceptual model for cross-domain microservices using event-driven and domain-driven design offers numerous benefits, including enhanced modularity, system resilience, and improved alignment between business and IT. However, these advantages come with significant challenges, such as the complexity of implementation and the need to handle eventual consistency (Bae & Park, 2014, Raza, 2021). Organizations must weigh these factors carefully, investing in the necessary expertise, tools, and processes to mitigate risks and maximize the model's potential. By doing so, they can create robust, scalable systems that meet the demands of today's dynamic and competitive environment.

### 2.6. Future Work

The development of a conceptual model for cross-domain microservices using event-driven and domain-driven design is a dynamic area of study that continues to evolve as new technologies and methodologies emerge. The foundational principles of modularity, decoupling, and domain alignment offer numerous advantages, but the future of this approach lies in further refining and enhancing its capabilities to meet the growing complexity of modern systems. One promising direction for future work involves exploring AI-driven monitoring and decision-making. This approach would integrate artificial intelligence into the core operations of microservices architectures, improving both the operational efficiency and adaptability of distributed systems (Asch, *et al*., 2018, Patel, *et al*., 2017).

AI-driven monitoring could enable intelligent analysis and anomaly detection within microservices environments. Traditional monitoring systems often rely on predefined thresholds and rule-based systems to identify issues, but these approaches may struggle to keep up with the dynamic nature of microservices, especially when systems scale or when new services are introduced (Wibowo, *et al*., 2017, Zheng, 2015). Integrating AI into the monitoring process would allow for real-time analysis of system behavior, offering more proactive detection of performance bottlenecks, service failures, or security breaches. Machine learning algorithms could be employed to detect patterns in system logs, network traffic, or event streams, identifying subtle anomalies that might otherwise go unnoticed by conventional systems. These AI models would improve as they are exposed to more data, learning from past incidents to predict and prevent future failures.

Moreover, AI could be applied to decision-making processes within microservices. For instance, AI models could assess the health of various services and automatically adjust service scaling or reallocate resources to maintain optimal system performance. This would go beyond traditional resource management strategies, which often require manual intervention, by introducing autonomous decision-making capabilities (Alessa, *et al*., 2016, Pace, Carpenter & Cole, 2015). AI could also play a crucial role in event-driven architectures, where events trigger automatic actions within the system. By applying machine learning models, systems could become more adaptive and self-healing, automatically reconfiguring based on the detected needs of the system. This approach would reduce the burden on developers and operations teams, enabling them to focus on higher-level strategic goals while the system itself autonomously handles routine adjustments.

The next frontier for extending cross-domain microservices models lies in integrating them with the Internet of Things (IoT) and other distributed systems. IoT devices are increasingly becoming integral parts of business operations across industries such as healthcare, manufacturing, transportation, and agriculture. These devices generate vast amounts of data and require sophisticated architectures to process, analyze, and act upon it in real-time. Event-driven and domain-driven design principles could be effectively applied to manage the data and actions generated by IoT devices (Bello, *et al*., 2023, Wulfert, *et al*., 2022). By treating IoT devices as services within a microservices architecture, each device could be managed as a separate bounded context with its own logic, allowing for scalable, efficient handling of the massive data streams they generate.

In such an extended model, IoT devices would produce events that are captured and processed by microservices. For instance, in a smart factory scenario, IoT sensors on production lines could trigger events related to equipment health, inventory levels, or production rates. These events would be captured by microservices that manage inventory, quality control, or maintenance. By using event-driven communication, the system could react in real-time to changes, such as adjusting the production schedule based on sensor data or alerting technicians to equipment malfunctions before they lead to downtime (Vlietland, Van Solingen & Van Vliet, 2016, Zhang, *et al*., 2017). This seamless integration between IoT and microservices would provide a unified platform for managing complex systems, where each service operates independently while still interacting with other services in a coherent manner.

Furthermore, the future development of this model would need to address the unique challenges posed by IoT and distributed systems. The sheer scale of IoT data, combined

with the need for low-latency processing, demands an architecture that can efficiently handle vast amounts of data while ensuring real-time responsiveness. Event-driven architectures are well-suited for this, as they allow systems to process events asynchronously, minimizing the time it takes to react to changes in the system (Duo, *et al*., 2022, Zong, 2022). However, challenges such as managing the consistency of data across distributed devices and microservices, ensuring the reliability of communication in environments with intermittent connectivity, and maintaining the security of sensitive data in IoT environments will require novel approaches.

In addition to IoT, distributed systems in general are becoming increasingly important in various industries, such as cloud computing, edge computing, and distributed databases. These systems often require complex coordination and communication between multiple services or nodes spread across different locations. By extending the cross-domain microservices model to these environments, organizations can leverage the benefits of modularity, resilience, and scalability to create highly distributed applications (Davis, 2014, Tang, Yilmaz & Cooke, 2018). For instance, in edge computing, where data processing occurs closer to the source of data generation, the integration of microservices can help manage the distribution of workloads and ensure that services at the edge work in harmony with those in the central cloud or data center.

The model's ability to integrate with distributed systems could also enhance the way businesses handle data privacy and security concerns. As data is processed across multiple nodes and services, especially in cloud or edge computing environments, maintaining privacy and security becomes increasingly challenging. Microservices architectures, by design, provide the flexibility to isolate sensitive data within specific services, applying stricter security measures as needed (Chen, *et al*., 2020, Saarikallio, 2022). This could include encryption of sensitive events, access control to ensure that only authorized services can access certain data, and auditing mechanisms to track data flows across the system. In the context of IoT, these capabilities are particularly important, as devices often collect highly sensitive information related to user behavior, health, or location.

In terms of scalability, integrating AI-driven monitoring and decision-making into microservices could also support the management of complex distributed systems by automating the scaling of services based on real-time demands. For example, if a sudden surge in device data from an IoT network is detected, AI could trigger an automatic increase in the resources allocated to the relevant services (Bitter, 2017, Rico, *et al*., 2018, Zou, *et al*., 2020). This would ensure that the system remains responsive, even during periods of high load. Similarly, in a cloud-based distributed system, AI could help manage the movement of workloads across different nodes, optimizing the use of resources while minimizing latency and downtime.

As organizations continue to adopt microservices, IoT, and distributed systems, the future of cross-domain microservices will increasingly involve an ecosystem of interconnected, intelligent services that can adapt to dynamic conditions. The combination of event-driven and domain-driven design principles, enhanced by AI-driven monitoring and IoT integration, offers a path toward more efficient, resilient, and scalable systems (Al-Ali, *et al*., 2016, Jones, *et al*., 2020). The development of these models will require ongoing research, experimentation, and collaboration across multiple disciplines, including software engineering, AI, and distributed computing. However, the potential to revolutionize the way businesses build and manage complex systems makes this area of work both exciting and essential for the future of digital transformation.

### 2.7. Conclusion

The development of a conceptual model for cross-domain microservices using event-driven and domain-driven design has proven to be a significant advancement in how distributed systems are structured and managed. By combining the strengths of both event-driven architectures and domain-driven design, this model allows for a more modular, scalable, and resilient approach to system design, especially in the context of complex, cross-domain applications. The contributions of this approach lie in its ability to decouple domains, ensuring that each microservice operates independently while still remaining cohesive within the larger ecosystem. This separation of concerns not only enhances system resilience but also facilitates the continuous evolution and improvement of individual domains without negatively impacting the overall system.

One of the key takeaways from this model is the emphasis on using events as the primary mechanism for communication between microservices. This approach allows for asynchronous processing, which is essential for maintaining responsiveness in large, distributed systems. Events trigger actions and responses across services, reducing the dependency between components and ensuring that changes in one domain are reflected in others without disrupting the entire architecture. This design paradigm also supports scalability, as services can be independently scaled based on demand, without the need for coordinated updates across the system. Additionally, by integrating domain-driven design, the model provides a clear structure for managing complex business logic and ensuring that each microservice is aligned with the specific requirements of its domain.

The implications for cross-domain microservices are far-reaching. The model demonstrates how organizations can create flexible, scalable systems that can adapt to changing business needs and technological advancements. By leveraging event-driven and domain-driven principles, companies can improve collaboration across domains, reduce system complexity, and increase the agility of their IT infrastructure. This is particularly important in industries such as e-commerce, financial services, and IoT, where business requirements and data flows are constantly evolving. The ability to quickly adapt and respond to new demands is crucial for maintaining a competitive edge.

Moreover, the conceptual model opens the door to further innovation. Future work, particularly in integrating AI-driven decision-making and extending the model to IoT and distributed systems, holds the promise of even more intelligent and responsive microservices architectures. As organizations continue to explore the full potential of this approach, the benefits will likely extend to greater operational efficiency, more personalized user experiences, and improved system security. Ultimately, the conceptual model for cross-domain microservices using event-driven and domain-driven design provides a solid foundation for the development of future-proof, high-performing systems that can meet the demands of an increasingly interconnected and

data-driven world.

## 3. References


1. Ahmad T, Aakula A, Ottori M, Saini V. Developing a strategic roadmap for digital transformation. J Comput Intell Robot. 2022;2(2):28-68.
2. Ahmaro I, Abualkishik AM, Yusoff MZM. Taxonomy, definition, approaches, benefits, reusability levels, factors and adaptation of software reusability: A review of the research literature. J Appl Sci. 2014;14(20):2396.
3. Al-Ali R, Kathiresan N, El Anbari M, Schendel ER, Zaid TA. Workflow optimization of performance and quality of service for bioinformatics application in high performance computing. J Comput Sci. 2016;15:3-10.
4. Alessa L, Kliskey A, Gamble J, Fidel M, Beaujean G, Gosz J. The role of Indigenous science and local knowledge in integrated observing systems: Moving toward adaptive capacity indices and early warning systems. Sustain Sci. 2016;11:91-102.
5. Ali N, Baker S, O'Crowley R, Herold S, Buckley J. Architecture consistency: State of the practice, challenges and requirements. Empir Softw Eng. 2018;23:224-58.
6. Andriyanto A, Doss R. X-driven methodologies for SOA system development—a survey. arXiv preprint. 2021;arXiv:2109.01805.
7. Asch M, Moore T, Badia R, Beck M, Beckman P, Bidot T, *et al*. Big data and extreme-scale computing: Pathways to convergence—toward a shaping strategy for a future software and data ecosystem for scientific inquiry. Int J High Perform Comput Appl. 2018;32(4):435-79.
8. Bae MJ, Park YS. Biological early warning system based on the responses of aquatic organisms to disturbances: A review. Sci Total Environ. 2014;466:635-49.
9. Bangemann T, Karnouskos S, Camp R, Carlsson O, Riedl M, McLeod S, *et al*. State of the art in industrial automation. In: Industrial Cloud-Based Cyber-Physical Systems: The IMC-AESOP Approach. 2014. p. 23-47.
10. Baxter C. Mastering Akka. Packt Publishing Ltd; 2016.
11. Becker T, Curry E, Jentzsch A, Palmetshofer W. New Horizons for a Data-Driven Economy: Roadmaps and Action Plans for Technology, Businesses, Policy, and Society. 2016.
12. Bellemare A. Building event-driven microservices. O'Reilly Media Inc.; 2020.
13. Bello OA, Folorunso A, Ejiofor OE, Budale FZ, Adebayo K, Babatunde OA. Machine learning approaches for enhancing fraud prevention in financial transactions. Int J Manag Technol. 2023;10(1):85-108.
14. Bello OA, Folorunso A, Ogundipe A, Kazeem O, Budale A, Zainab F, Ejiofor OE. Enhancing cyber financial fraud detection using deep learning techniques: A study on neural networks and anomaly detection. Int J Net Commun Res. 2022;7(1):90-113.
15. Bello OA, Folorunso A, Onwuchekwa J, Ejiofor OE. A comprehensive framework for strengthening USA financial cybersecurity: Integrating machine learning and AI in fraud detection systems. Eur J Comput Sci Inf Technol. 2023;11(6):62-83.
16. Bello OA, Folorunso A, Onwuchekwa J, Ejiofor OE, Budale FZ, Egwuonwu MN. Analysing the impact of advanced analytics on fraud detection: A machine learning perspective. Eur J Comput Sci Inf Technol. 2023;11(6):103-26.
17. Bhaskaran SV. Integrating data quality services (DQS) in big data ecosystems: Challenges, best practices, and opportunities for decision-making. J Appl Big Data Anal Decis-Making Predictive Modelling Syst. 2020;4(11):1-12.
18. Bitter J. Improving multidisciplinary teamwork in preoperative scheduling. [dissertation]. [Place of publication]: [Publisher]; 2017.
19. Cerqueus A, Delorme X. Evaluating the scalability of reconfigurable manufacturing systems at the design phase. Int J Prod Res. 2023;61(23):8080-93.
20. Chen Q, Hall DM, Adey BT, Haas CT. Identifying enablers for coordination across construction supply chain processes: A systematic literature review. Eng Constr Archit Manag. 2020;28(4):1083-1113.
21. Chinamanagonda S. Observability in Microservices Architectures-Advanced observability tools for microservices environments. MZ Comput J. 2022;3(1).
22. Classi CC, Nowicki DR, Mansouri M, Sauser BJ, Randall WS. A systems thinking approach to managing sustainment phase redesign planning. Eng Manag J. 2018;30(1):68-81.
23. Curuksu JD. Data driven. Manag for Prof. 2018.
24. Dal Maso A. The evolution of Data Governance: a tool for an improved and enhanced decision-making process. 2019.
25. Dalibor M, Jansen N, Rumpe B, Schmalzing D, Wachtmeister L, Wimmer M, *et al*. A cross-domain systematic mapping study on software engineering for digital twins. J Syst Softw. 2022;193:111361.
26. Datta SK, Bonnet C. Next-generation, data centric and end-to-end IoT architecture based on microservices. In: 2018 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia); 2018 Jun 11-13; Seoul, South Korea. IEEE; 2018. p. 206-12.
27. Davis JE. Temporal meta-model framework for Enterprise Information Systems (EIS) development [dissertation]. Curtin University; 2014.
28. Debski A, Szczepanik B, Malawski M, Spahr S, Muthig D. In search for a scalable & reactive architecture of a cloud application: CQRS and event sourcing case study. IEEE Softw. 2017;99.
29. Diatte K, O'Halloran B, Van Bossuyt DL. The integration of reliability, availability, and maintainability into model-based systems engineering. Syst. 2022;10(4):101.
30. Dobaj J, Iber J, Krisper M, Kreiner C. A microservice architecture for the industrial Internet-of-Things. In: Proceedings of the 23rd European Conference on Pattern Languages of Programs; 2018 Jul 2-4; Irsee, Germany. p. 1-15.
31. Dulam N, Katari A, Allam K. Data Mesh in Practice: How Organizations Are Decentralizing Data Ownership. Distribut Learn Broad Appl Sci Res. 2020;6.
32. Duo X, Xu P, Zhang Z, Chai S, Xia R, Zong Z. KCL: A Declarative Language for Large-Scale Configuration and Policy Management. In: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications; 2022 Oct 6-8; Nanjing, China. Cham: Springer Nature Switzerland; 2022. p. 88-105.
33. Ehikioya SA, Guillemot E. A critical assessment of the design issues in e-commerce systems development. Eng Rep. 2020;2(4):e12154.

34. Elouataoui W, El Alaoui I, El Mendili S, Gahi Y. An advanced big data quality framework based on weighted metrics. Big Data Cogn Comput. 2022;6(4):153.
35. Elujide I, Fashoto SG, Fashoto B, Mbunge E, Folorunso SO, Olamijuwon JO. Application of deep and machine learning techniques for multi-label classification performance on psychotic disorder diseases. Informatics Med Unlocked. 2021;23:100545.
36. Elujide I, Fashoto SG, Fashoto B, Mbunge E, Folorunso SO, Olamijuwon JO. Informatics in Medicine Unlocked. 2021.
37. Fedushko S, Ustyianovych T. E-commerce customers behavior research using cohort analysis: A case study of COVID-19. J Open Innov Technol Mark Comple. 2022;8(1):12.
38. Gade KR. Migrations: Cloud Migration Strategies, Data Migration Challenges, and Legacy System Modernization. J Comput Inform Technol. 2021;1(1).
39. Gökalp MO, Gökalp E, Kayabay K, Koçyiğit A, Eren PE. Data-driven manufacturing: An assessment model for data science maturity. J Manuf Syst. 2021;60:527-46.
40. Guzmán GI, González de Molina M. Energy efficiency in agrarian systems from an agroecological perspective. Agroecol Sustain Food Syst. 2015;39(8):924-52.
41. Hahn J, Lee G. The complex effects of cross-domain knowledge on IS development: A simulation-based theory development. MIS Q. 2021;45(4):2023-54.
42. Hashem IAT, Yaqoob I, Anuar NB, Mokhtar S, Gani A, Khan SU. The rise of "big data" on cloud computing: Review and open research issues. Inf Syst. 2015;47:98-115.
43. Hassan M, Gregory MA, Li S. Multi-Domain Federation Utilizing Software Defined Networking—A Review. IEEE Access. 2023;11:19202-27.
44. Intizar M, Patel P, Datta SK, Gyrard A. Multi-layer cross domain reasoning over distributed autonomous IoT applications. Open J Internet Things. 2017;3.
45. Jones CL, Golanz B, Draper GT, Janusz P. Practical Software and Systems Measurement Continuous Iterative Development Measurement Framework. Version 1. 2020;15.
46. Jones SC. Impact & excellence: data-driven strategies for aligning mission, culture and performance in nonprofit and government organizations. Hoboken: John Wiley & Sons; 2014.
47. Joseph PT. E-commerce: An Indian perspective. New Delhi: PHI Learning Pvt. Ltd.; 2023.
48. Kalske M. Transforming monolithic architecture towards microservice architecture. University of Helsinki; 2017.
49. Kayabay K, Gökalp MO, Gökalp E, Eren PE, Koçyiğit A. Data science roadmapping: An architectural framework for facilitating transformation towards a data-driven organization. Technol Forecast Soc Chang. 2022;174:121264.
50. Kempa EE, Smith CA, Li X, Bellina B, Richardson K, Pringle S, *et al*. Coupling droplet microfluidics with mass spectrometry for ultrahigh-throughput analysis of complex mixtures up to and above 30 Hz. Anal Chem. 2020;92(18):12605-12.
51. Khan A. Microservices in context: Internet of Things: Infrastructure and Architecture. 2017.
52. Korkmaz N, Nilsson M. Practitioners' view on command query responsibility segregation. MSc Thesis. 2014.
53. Labiadh M, Obrecht C, Ferreira da Silva C, Ghodous P. A microservice-based framework for exploring data selection in cross-building knowledge transfer. Serv Oriented Comput Appl. 2021;15:97-107.
54. Laranjeiro N, Soydemir SN, Bernardino J. A survey on data quality: classifying poor data. In: 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC); 2015 Nov 30-Dec 3; Fukuoka, Japan. IEEE; 2015. p. 179-88.
55. Macero M, Macero A, Anglin. Learn Microservices with Spring Boot. Berkeley: Apress; 2017.
56. Maja MM, Letaba P. Towards a data-driven technology roadmap for the bank of the future: Exploring big data analytics to support technology roadmapping. Soc Sci Humanit Open. 2022;6(1):100270.
57. Malik A, Qadir J, Ahmad B, Yau KLA, Ullah U. QoS in IEEE 802.11-based wireless networks: A contemporary review. J Netw Comput Appl. 2015;55:24-46.
58. Martinetti A, Schakel EJ, van Dongen LA. Flying asset: Framework for developing scalable maintenance program for unmanned aircraft systems (UAS). J Qual Maint Eng. 2018;24(2):152-69.
59. Mishra S, Komandla V, Bandi S. A domain-driven data architecture for improving data quality in distributed datasets. J Artif Intell Res Appl. 2021;1(2):510-31.
60. Mishra S, Komandla V, Bandi S. A new pattern for managing massive datasets in the enterprise through Data Fabric and Data Mesh. J AI-Assisted Sci Discov. 2021;1(2):236-59.
61. Morrell WC, Birkel GW, Forrer M, Lopez T, Backman TW, Dussault M, *et al*. The experiment data depot: A web-based software tool for biological experimental data storage, sharing, and visualization. ACS Synth Biol. 2017;6(12):2248-59.
62. Mustalahti I, Rakotonarivo OS. REDD+ and empowered deliberative democracy: Learning from Tanzania. World Dev. 2014;59:199-211.
63. N'guessan GB, Achiepo OY, Diako J. Event Sourcing and Command Query Responsibility Segregation Based Fast Data Architecture. Open J Appl Sci. 2023;13(2):198-206.
64. Nalla LN, Reddy VM. Scalable data storage solutions for high-volume e-commerce transactions. Int J Adv Eng Technol Innov. 2021;1(4):1-16.
65. Navarro LFM. Investigating the influence of data analytics on content lifecycle management for maximizing resource efficiency and audience impact. J Comput Soc Dyn. 2017;2(2):1-22.
66. Newman S. Monolith to microservices: evolutionary patterns to transform your monolith. O'Reilly Media; 2019.
67. Nguyen T. Elementary event storage. 2019.
68. Nilsson M, Korkmaz N. Practitioners' view on command query responsibility segregation. 2014.
69. Nimmagadda VSP. Artificial intelligence for supply chain visibility and transparency in retail: advanced techniques, models, and real-world case studies. J Mach Learn Pharm Res. 2023;3(1):87-120.
70. Özkan O, Babur Ö, Brand MV. Domain-driven design in software development: a systematic literature review on implementation, challenges, and effectiveness. arXiv preprint arXiv:2310.01905. 2023.
71. Pace ML, Carpenter SR, Cole JJ. With and without warning: managing ecosystems in a changing world.

Front Ecol Environ. 2015;13(9):460-467.
72. Patel A, Alhussian H, Pedersen JM, Bounabat B, Júnior JC, Katsikas S. A nifty collaborative intrusion detection and prevention architecture for smart grid ecosystems. Comput Secur. 2017;64:92-109.
73. Peng G, Privette JL, Kearns EJ, Ritchey NA, Ansari S. A unified framework for measuring stewardship practices applied to digital environmental datasets. Data Sci J. 2015;13:231-253.
74. Pestana JS. Data governance valuation: a model for assessing the impact on organisations' business. Master's thesis, Universidade NOVA de Lisboa, Portugal; 2023.
75. Pora U, Gerdsri N, Thawesaengskulthai N, Triukose S. Data-driven roadmapping (DDRM): approach and case demonstration. IEEE Trans Eng Manag. 2020;69(1):209-227.
76. Pora U, Thawesaengskulthai N, Gerdsri N, Triukose S. Data-driven roadmapping turning challenges into opportunities. In: 2018 Portland International Conference on Management of Engineering and Technology (PICMET); 2018 Aug; Portland, OR, USA. IEEE; 2018. p. 1-11.
77. Prasat K, Sanjay S, Ananya V, Kannadasan R, Rajkumar S, Raut R, Selvanambi R. Analysis of cross-domain security and privacy aspects of cyber-physical systems. Int J Wirel Inf Netw. 2022;29(4):454-479.
78. Pulwarty RS, Sivakumar MV. Information systems in a changing climate: early warnings and drought risk management. Weather Clim Extremes. 2014;3:14-21.
79. Pyykkö H, Suoheimo M, Walter S. Approaching sustainability transition in supply chains as a wicked problem: systematic literature review in light of the evolved double diamond design process model. Processes. 2021;9(12):2135.
80. Rafique W, Khan M, Dou W. Maintainable software solution development using collaboration between architecture and requirements in heterogeneous IoT paradigm (Short Paper). In: Collaborative Computing: Networking, Applications and Worksharing: 15th EAI International Conference, CollaborateCom 2019, London, UK, August 19-22, 2019, Proceedings 15; 2019. Springer International Publishing. p. 489-508.
81. Raza H. Proactive cyber defense with AI: enhancing risk assessment and threat detection in cybersecurity ecosystems. 2021.
82. Razzaq A, Ghayyur SA. A systematic mapping study: the new age of software architecture from monolithic to microservice architecture—awareness and challenges. Comput Appl Eng Educ. 2023;31(2):421-451.
83. Ribeiro RJ. Safe API evolution in a microservice architecture with a pluggable and transactionless solution. 2022.
84. Rico R, Hinsz VB, Davison RB, Salas E. Structural influences upon coordination and performance in multiteam systems. Hum Resour Manag Rev. 2018;28(4):332-346.
85. Rishaug R, Wika OL. Event sourcing. Bachelor's thesis, NTNU; 2018.
86. Saarikallio M. Improving hybrid software business: quality culture, cycle-time, and multi-team agile management. JYU Dissertations; 2022.
87. Sabou M, Biffl S, Einfalt A, Krammer L, Kastner W, Ekaputra FJ. Semantics for cyber-physical systems: a cross-domain perspective. Semantic Web. 2020;11(1):115-124.
88. Saidi M, Tissaoui A, Faiz S. A DDD approach towards automatic migration to microservices. In: 2023 IEEE International Conference on Advanced Systems and Emergent Technologies (IC_ASET); 2023 Apr; 2023. p. 01-06.
89. Saiod AK, Van Greunen D, Veldsman A. Electronic health records: benefits and challenges for data quality. In: Handbook of Large-Scale Distributed Computing in Smart Healthcare. 2017. p. 123-156.
90. Shrestha YR, Krishna V, von Krogh G. Augmenting organizational decision-making with deep learning algorithms: principles, promises, and challenges. J Bus Res. 2021;123:588-603.
91. Siddiqa A, Hashem IAT, Yaqoob I, Marjani M, Shamshirband S, Gani A, Nasaruddin F. A survey of big data management: taxonomy and state-of-the-art. J Netw Comput Appl. 2016;71:151-166.
92. Solberg E. The transition from monolithic architecture to microservice architecture: a case study of a large Scandinavian financial institution. Master's thesis, 2022.
93. Strengholt P. Data management at scale. O'Reilly Media, Inc.; 2023.
94. Sturtevant C, DeRego E, Metzger S, Ayres E, Allen D, Burlingame T, SanClements M. A process approach to quality management doubles NEON sensor data quality. Methods Ecol Evol. 2022;13(9):1849-1865.
95. Tang P, Yilmaz A, Cooke N. Automatic imagery data analysis for proactive computer-based workflow management during nuclear power plant outages. No. 15-8121, Arizona State University, Tempe, AZ (USA); 2018.
96. Tapia F, Mora MA, Fuertes W, Aules H, Flores E, Toulkeridis T. From monolithic systems to microservices: a comparative study of performance. Appl Sci. 2020;10(17):5797.
97. Tchernykh A, Schwiegelsohn U, Talbi EG, Babenko M. Towards understanding uncertainty in cloud computing with risks of confidentiality, integrity, and availability. J Comput Sci. 2019;36:100581.
98. Tidjon LN, Frappier M, Mammar A. Intrusion detection systems: a cross-domain overview. IEEE Commun Surv Tutor. 2019;21(4):3639-3681.
99. Vallejo-Vaz AJ, Akram A, Seshasai SRK, Cole D, Watts GF, Hovingh GK, EAS Familial Hypercholesterolaemia Studies Collaboration. Pooling and expanding registries of familial hypercholesterolaemia to assess gaps in care and improve disease management and outcomes: rationale and design of the global EAS Familial Hypercholesterolaemia Studies Collaboration. Atheroscler Suppl. 2016;22:1-32.
100. Vernon V, Jaskula T. Strategic monoliths and microservices: driving innovation using purposeful architecture. 2021.
101. Vlietland J, Van Solingen R, Van Vliet H. Aligning codependent Scrum teams to enable fast business value delivery: a governance framework and set of intervention actions. J Syst Softw. 2016;113:418-429.
102. Weiler JR, Lomotey K. Defining rigor in justice-oriented EdD programs: preparing leaders to disrupt and transform schools. Educ Adm Quart. 2022;58(1):110-140.
103. Wibowo FX, Gregory MA, Ahmed K, Gomez KM.

Multi-domain software defined networking: research status and challenges. J Netw Comput Appl. 2017;87:32-45.
104. Wulfert T, Woroch R, Strobel G, Seufert S, Möller F. Developing design principles to standardize e-commerce ecosystems: a systematic literature review and multi-case study of boundary resources. Electron Mark. 2022;32(4):1813-1842.
105. Yu W, Dillon T, Mostafa F, Rahayu W, Liu Y. A global manufacturing big data ecosystem for fault detection in predictive maintenance. IEEE Trans Ind Informatics. 2019;16(1):183-192.
106. Zhang C, Tang P, Cooke N, Buchanan V, Yilmaz A, Germain SWS, Gupta A. 2023.