INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Integration of machine learning models in a microservices architecture

Ahmed Gamal Ibrahim - a53168

Thesis presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Rui Pedro Lopes

This document does not include the suggestions made by the board.

Bragança

2023-2024

Integration of machine learning models in a microservices architecture

Ahmed Gamal Ibrahim - a53168

Thesis presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Rui Pedro Lopes

This document does not include the suggestions made by the board.

Bragança

2023-2024

Dedication

To my family, whose unwavering support has been my foundation and my motivation through every step of this journey.

And to my supervisor for his invaluable guidance, motivation, patience, and wisdom, which have pushed me forward and inspired me to keep going forward with dedication.

Acknowledgment

This work would not have been possible without the support of the OpenZDM project. I am sincerely grateful to the entire team for their collaboration, resources, and expertise, which have been instrumental in advancing my understanding and contributions to Zero Defect Manufacturing. The knowledge and insights shared by my colleagues within the project provided a strong foundation for this research, and I am deeply appreciative of the opportunity to work alongside such dedicated professionals.

Abstract

Achieving Zero Defect Manufacturing in the evolving landscape of Industry 4.0 requires advanced, scalable architectures that support proactive quality management and real-time defect detection. This thesis introduces a ZDM-focused microservices architecture designed to enhance modularity, resilience, and scalability within industrial manufacturing environments. By integrating Cyber-Physical Systems and Digital Twins, the proposed architecture facilitates continuous monitoring, dynamic data flow, and predictive analytics, aligning with RAMI 4.0 standards to ensure seamless interoperability across systems.

Emphasizing communication-driven design, the architecture leverages distributed microservices and specialized communication brokers to create a flexible, event-driven system. This enables efficient handling of high data volumes, real-time quality insights, and early anomaly detection. Through a structured evaluation of core architectural components, including orchestration, choreography, and communication brokers, this work establishes a foundation for ZDM implementations adaptable to various industrial settings.

The architecture's deployment in a real-world manufacturing case demonstrates its practical benefits, illustrating how modular, scalable systems can drive operational improvements and defect reduction. Contributing to the broader Industry 4.0 framework, this work provides a blueprint for future ZDM solutions that prioritize sustainability, adaptability, and enhanced product quality in complex manufacturing ecosystems.

Keywords: Microservices architecture, Message brokers, Industry 4.0

Contents

List of Figures

Acronyms

AAA Authentication, Authorization and Accounting.

AAS Asset Administration Shell.

API Application Programming Interface.

CMS Content Management System.

CPS Cyper-Physical Systems.

CQRS command query responsibility segregation.

DAO Data Access Object.

DB Data Base.

DT Digital Twin.

EDA Event-Driven Architecture.

ESTiG Escola Superior de Tecnologia e Gestão.

HTTP HyperText Transfer Protocol.

I4.0 Industry 4.0.

I5.0 Industry 5.0.

IoT Internet of Things.

IPB Instituto Politécnico de Bragança.

LHC Large Hadron Collider.

MOM Messaging-Oriented Middleware.

NDI Non-Destructive Inspection.

RAMI4.0 Reference Architecture Model for I4.0.

RBAC Role-Based Access Control.

SOA Service-Oriented Srchitectures.

XaaS Everything-as-a-Service.

XSD XML Schema Definition.

ZDM Zero Defect Manufacturing.

Chapter 1

Introduction

In the era of Industry 4.0 (I4.0), the global manufacturing landscape is undergoing a profound transformation driven by digitalization and the integration of advanced technologies such as Cyper-Physical Systems (CPS), Digital Twin (DT), and Internet of Things (IoT). These innovations enable the shift from traditional, reactive manufacturing approaches to more predictive and prescriptive models, leading to the emergence of concepts like ZDM. ZDM is designed to eliminate defects by identifying and addressing issues early in the production process, thus enhancing quality, reducing waste, and increasing efficiency. With the advent of Industry 5.0 (I5.0), there is a growing emphasis on human-centered, resilient, and sustainable manufacturing practices, making ZDM even more critical for modern industries.

At the core of ZDM is the ability to harness vast amounts of data generated across manufacturing systems, analyze it in real-time, and make informed decisions to prevent defects. This demand for real-time data processing, high scalability, and fault tolerance has driven the adoption of microservices architecture in industrial applications. Microservices decompose monolithic systems into smaller, independent services that communicate with each other through well-defined APIs. This modular approach allows for greater flexibility, scalability, and fault tolerance, making it an ideal solution for managing the complexity of modern industrial systems.

1.1 Objectives

This thesis focuses on the integration of microservices architecture with communication brokers to support ZDM within an I4.0 framework. Specifically, it explores how communication brokers such as Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS can facilitate efficient, real-time data exchange between distributed microservices in a ZDM environment. By enabling asynchronous, loosely coupled interactions, these brokers play a pivotal role in managing data flows and ensuring the reliability and scalability of industrial systems.

The motivation behind this research stems from the growing complexity of manufacturing environments, where the need for real-time monitoring and predictive quality control is becoming more apparent. Traditional monolithic architectures are no longer sufficient to handle the dynamic, data-intensive demands of modern production lines. Microservices, combined with robust communication brokers, offer a solution to this challenge by enabling flexible, scalable, and fault-tolerant systems capable of processing large volumes of data in real-time.

The objectives of this research are threefold:

1. To explore the evolution of microservices architecture and its applicability to ZDM in industrial settings.

2. To evaluate the performance of different communication brokers (Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS) in supporting real-time data exchange between microservices in a ZDM framework.

3. To implement and demonstrate the effectiveness of a microservices-based ZDM architecture in a practical industrial case study, highlighting its scalability, flexibility, and ability to minimize defects in manufacturing.

1.2 Structure of the Document

The structure of this thesis is as follows:

- **Chapter 2** provides a comprehensive review of the state-of-the-art technologies underpinning ZDM and microservices architectures, including CPS, DT, communication brokers, and the transition from monolithic to distributed systems. This chapter also explores the evolution of microservices in the context of I4.0 and I5.0.
- **Chapter 3** delves into the communication and coordination mechanisms of microservices, focusing on the role of communication brokers. It presents an in-depth comparison of Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS, evaluating their performance in various real-time messaging scenarios and highlighting their suitability for ZDM applications.
- **Chapter 4** introduces a microservices-based architecture designed to support ZDM, detailing its implementation in an industrial case study. This chapter covers the data pipeline, communication patterns, and real-time analytics used to detect and prevent defects in a production environment.
- **Chapter 5** concludes the thesis by summarizing the key findings and contributions of the research, discussing the challenges faced during implementation, and proposing directions for future research in the domain of ZDM and microservices in manufacturing.

Through this research, the thesis aims to contribute to the growing body of knowledge in ZDM and microservices architecture by demonstrating the practical benefits of integrating these technologies in modern manufacturing environments. By leveraging communication brokers to enhance the coordination and scalability of microservices, this work offers a scalable, flexible, and fault-tolerant solution to the challenges of defect reduction in the manufacturing industry.

Chapter 2

State of the art

ZDM and the evolution of microservices architectures have become critical in modern industrial systems and software development. Both concepts strive for efficiency, scalability, and resilience and are deeply intertwined with advanced technologies such as CPS, DT, communication brokers, and Service-Oriented Srchitectures (SOA). This section will explore these concepts by reviewing the relevant literature, highlighting the technological advancements and challenges associated with each, while also providing a comprehensive overview of key tools and frameworks enabling ZDM and microservices architectures in I4.0 and I5.0.

2.1 ZDM and Digital Twins

ZDM and DT have progressively shaped modern manufacturing, aiming to increase quality and operational resilience by minimizing or ideally eliminating product defects. This shift from conventional quality control, based on detection and correction, to predictive and preventive strategies, is bolstered by advancements in digital technology under I4.0 framework. Early ZDM efforts can be traced back to the quality-first philosophies that arose in the 1960s, most notably with Toyota's Zero Defects Campaign and Crosby's concept of "zero defects" as a quality standard [1]. The digitalization surge in the 2010s introduced capabilities that allowed for real-time monitoring and analytics, crucial for

achieving ZDM at scale.

The integration of DT enhances ZDM by creating real-time, data-driven representations of physical assets, enabling continuous monitoring, simulation, and optimization of manufacturing processes. DT provide manufacturers with predictive insights, allowing them to proactively prevent defects and improve product quality. In this context, DT technology supports ZDM not only in defect prediction but also in compensation strategies that align with the dynamic, multi-stage nature of modern production systems, making it a critical enabler of I5.0 human-centered approach [1].

For comprehensive ZDM deployment, frameworks like Reference Architecture Model for I4.0 (RAMI4.0) provide structured guidelines that integrate DT, AI, and IoT technologies, allowing seamless data flow across production layers. As manufacturing shifts towards I5.0, the role of DT and ZDM expands to emphasize sustainability, resilience, and circular economy principles—integrating quality control with waste minimization and resource efficiency strategies, a research area still rich with potential developments [1].

The concept of ZDM revolves around the minimization of defective products through the integration of digitalization and advanced analytics throughout the entire value chain of production. Several researchers have explored architectures to enable ZDM, focusing on the interconnection between physical processes and their digital counterparts. A key framework proposed by Konstantinidis et al. [2] presents a layered architecture for achieving ZDM in dairy production, consisting of four layers: the physical layer, the vision-based layer, the DT layer, and the ZDM layer. This architecture integrates real-time data collection, predictive analytics, and quality control to ensure that defects are minimized and that maintenance costs are reduced through predictive maintenance strategies.

Similarly, Magnanini et al. [3] proposed a multi-layered system where data from shop-floor operations is gathered, synchronized, and optimized for quality control and production logistics. These studies underscore the critical role of digital twins in simulating and optimizing production processes, thereby enabling proactive quality control measures. The integration of DT in ZDM frameworks, as described by Martinez et al. [4], allows for product inspection, machine health analysis, and preventive maintenance,

ensuring that defects are detected and resolved before products leave the factory floor.

ZDM is further enhanced through Non-Destructive Inspection (NDI) methods, as outlined by Medici et al. [5]. NDI techniques are fundamental in digitizing physical processes for data acquisition, storage, and analysis, which subsequently feed into decision-making systems for quality control. The use of I4.0 standards such as RAMI4.0 also helps provide a structured approach for managing the complexity of industrial CPS. Sølvsberg et al. [6] focus on the design and development of a data structure within the Asset Administration Shell (AAS) in RAMI4.0, allowing for efficient multivariate analysis and scalability of industrial CPS.

The move toward I5.0 further emphasizes human-centered and resilient production systems. According to Zeb et al. [7], the focus in I5.0 shifts to personalized services, supported by AI-native service architectures and high data-transportation networks. The concept of Everything-as-a-Service (XaaS) plays a significant role in this paradigm, as it facilitates rapid tool provisioning and real-time analysis critical for ZDM systems. The flexibility of these SOA enables enterprises to evolve their manufacturing processes continuously, aligning with both I4.0 and I5.0 visions.

2.2 Microservices Architecture and Communication Brokers

The evolution of software architectures has been driven by the increasing demand for scalable, efficient, and reliable systems, particularly in the context of modern industrial and large-scale applications. At the core of this evolution is the concept of distributed systems, which allow multiple computing devices to work together, sharing resources and processing tasks across networks.

Distributed systems have been an active field of research for over 60 years, playing a crucial role in Computer Science by enabling the invention of the Internet, which underpins all facets of modern life. These systems have evolved from early mainframe machines

that provided centralized computing and storage interfaced by teletype terminals, to the complex paradigms we observe today, such as cloud computing, fog computing, and IoT [8]. Moreover, it eventually led to Distributed Computing Continuum Systems that unify cloud, edge, IoT, and mobile devices into a seamless continuum [9]. Initially, distributed systems emerged as businesses and industries sought to overcome the limitations of centralized, monolithic computing models. Early examples include client-server architectures, which enabled users to interact with centralized servers via clients, distributing the load and allowing for greater flexibility. As these systems evolved, the need for more decoupled, modular, and independently scalable services became apparent.

Monolithic architectures, where all components of an application are built into a single, unified system, initially dominated software development. However, as applications grew in complexity, this model revealed significant limitations:

- **Scalability issues**: Monolithic systems required scaling the entire application even if only one part needed more resources.
- **Complexity in updates**: Any modification or update, no matter how small, necessitated redeployment of the entire application, increasing the risk of errors and downtime.
- **Lack of flexibility**: The tightly-coupled nature of monolithic systems made it difficult to adopt new technologies or frameworks for individual parts of the application without affecting the entire system.

To address these challenges, the industry shifted toward distributed architectures, which spread processing tasks across multiple nodes or systems, each performing specific functions. SOA was one such model that gained prominence in the early 2000s, promoting the development of modular, reusable services that could communicate with each other via standardized interfaces like web services.

While SOA brought significant improvements, it still relied on complex middleware solutions such as the Enterprise Service Bus (ESB) for message routing, which became a

bottleneck in large-scale, dynamic environments. This is where microservices architecture emerged as a more agile, decentralized alternative.

Microservices architecture builds upon SOA principles but introduces several key innovations:

- **Decentralized data management**: Each microservice manages its own data and database, which eliminates shared data dependencies and reduces bottlenecks.
- **Service autonomy**: Microservices are independent, meaning they can be developed, deployed, and scaled individually without affecting other services in the system.
- **Lightweight communication**: Unlike SOA heavy reliance on ESB, microservices typically use lightweight communication protocols like HTTP/REST, gRPC, or event-driven architectures to allow services to interact in a loosely-coupled manner.

By decomposing applications into small, focused services, microservices architecture enables greater scalability, fault tolerance, and the ability to adopt different technologies for each service as needed.

SOA was developed in the early 2000s to address challenges inherent in monolithic architectures. SOA structures applications into modular, self-contained services that perform specific business functions. Each service in SOA can communicate with other services through a common interface, promoting reuse and modularity across applications. However, SOA reliance on centralized data and an Enterprise Service Bus (ESB) for message routing can increase complexity, especially when scaling in dynamic environments [10], [11]. Despite these limitations, SOA marked a transformative shift by decoupling services and promoting interoperability, which laid the groundwork for cloud-native development [12], [13].

Microservices architecture builds upon SOA principles but decentralizes data and governance, optimizing services for scalability and independence. In microservices, each service operates autonomously with its own database and is managed by a dedicated team.

This architecture enables more frequent, smaller deployments and rapid scaling, as each microservice can be independently updated or replaced without affecting the entire system [10], [14]. By promoting "bounded contexts," where services manage only their internal data, microservices avoid shared dependencies common in SOA, allowing more agile responses to evolving business needs [11], [12].

2.2.1 Orchestration vs. Choreography

To manage inter-service communication in microservices, two main patterns are used: choreography and orchestration.

Choreography is a decentralized, event-driven model where services communicate asynchronously by publishing and subscribing to events. This pattern enhances service autonomy and scalability by allowing services to respond independently to changes without relying on a central controller. Choreography is ideal for environments that prioritize loose coupling and scalability, though it requires careful design to ensure data consistency and handle potential failure scenarios [11], [14], [15]. A concrete example of event choreography can be found in microservices that execute distributed transactions using the Saga pattern. For instance, in an e-commerce application, placing an order triggers events that sequentially update the order status, deduct inventory, and charge the customer's credit card. Each service publishes events that other services subscribe to, without a central controller. If one of the services fails, the system triggers compensating transactions in reverse order to roll back the process. This method demonstrated faster performance in scenarios involving fewer services and less complexity, where communication overhead is minimized and decentralization offers flexibility [16].

In addition to flexibility, choreography promotes high **scalability and fault tolerance** by enabling each service to operate independently. The absence of a central controller reduces single points of failure, and high-throughput messaging platforms, such as Kafka or RabbitMQ, facilitate these asynchronous interactions. This makes it possible to process high transaction volumes and scale each service individually based on workload

demands, which is particularly beneficial in large, dynamic environments.

One real-world example of choreography in action is in an e-commerce order fulfillment system, where autonomous services such as inventory management, payment processing, and shipping operate independently based on events, allowing them to process tasks asynchronously while adapting to high-volume traffic. This loose coupling between services also reduces dependencies, allowing faster and more flexible deployment.

However, the decentralized nature of choreography presents challenges, especially in **error handling and observability**. Because each service independently manages its own error recovery, the compensation logic is scattered across the architecture, making it more complex to design, monitor, and maintain than a centralized orchestrator approach. Distributed tracing tools like Jaeger or OpenTelemetry are essential in this context to provide visibility into the system's event flow, helping address issues like "event storming" where high event traffic could potentially overwhelm the system.

Compensation and Error Handling: When a failure occurs in a choreographed setup, each service must independently handle error scenarios through compensating events to ensure data consistency. This decentralized error handling promotes resilience by preventing cascading failures but requires careful planning to ensure that compensations are applied consistently across all affected services.

Orchestration involves a central controller that dictates the interactions between services. This model is well-suited for workflows requiring complex, sequential tasks and centralized management. However, orchestration can introduce dependencies that hinder the autonomous nature of microservices, making it more suitable for controlled environments [14], [15].

In orchestration, the central controller (orchestrator) explicitly defines the order of operations, allowing for a clear and structured workflow that ensures each service is called in a specific sequence. This approach is particularly useful in scenarios requiring strong control over distributed transactions, as the orchestrator can coordinate each step and centralize the error-handling process. For example, in an e-commerce application, orchestration would involve a central orchestrator service that monitors events and decides

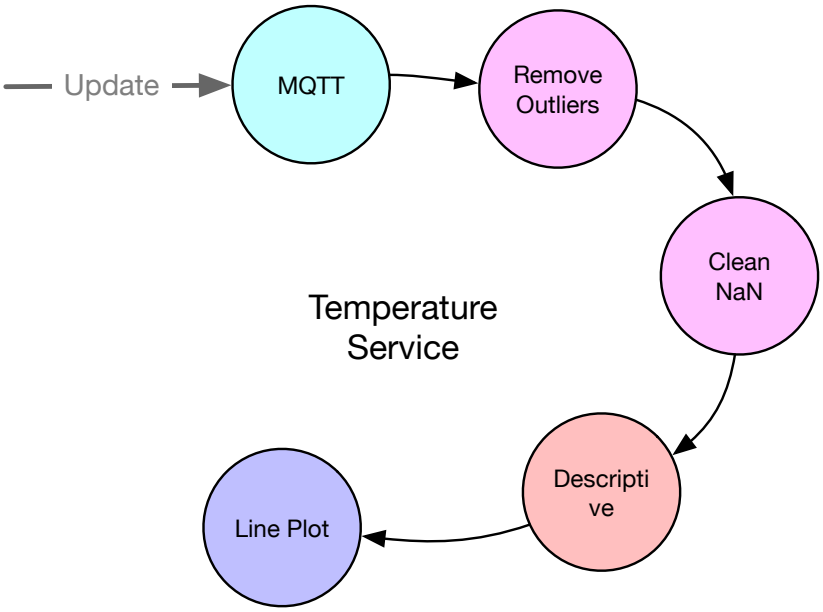

Figure 2.1: Choreography

which service to trigger next. When an order is placed, the orchestrator manages the flow of events—billing, shipping, and inventory updates—ensuring that tasks are completed in the correct sequence [16].

A significant benefit of orchestration is its centralized control, which allows for greater oversight and easier debugging. The orchestrator acts as a single point of monitoring, making it straightforward to trace the sequence of operations and diagnose issues as they arise. This centralized structure simplifies the handling of complex workflows by ensuring that error recovery steps, such as rolling back transactions, are managed consistently across the entire workflow.

However, the centralized nature of orchestration also introduces certain challenges. The orchestrator can become a single point of failure in the system, which may impact the overall reliability of the architecture. Additionally, because services are dependent on the orchestrator, the architecture may experience reduced flexibility and scalability compared to a decentralized model. In high-traffic environments, the orchestrator can become a bottleneck, potentially slowing down performance if it cannot manage the load efficiently.

Orchestration also leads to tighter coupling among services, as each service must interact directly with the orchestrator. This close interaction can limit the ability to independently update or replace services without modifying the central orchestrator, which may complicate maintenance and increase the cost of implementing changes over time.

Compensation and Error Handling: A key advantage of orchestration is its centralized error handling. If an error occurs during a transaction, the orchestrator can initiate compensating transactions to roll back previous successful steps, ensuring that the system remains consistent. This centralized approach to error handling makes orchestration particularly suited for processes with complex error recovery requirements, where consistent rollback across services is essential to maintain data integrity.

Design Considerations: While orchestration provides structure, it is crucial to consider the potential limitations on scalability and resilience. For large-scale applications, designing the orchestrator to handle distributed workloads efficiently is essential. Additionally, implementing redundancy or failover mechanisms can help mitigate the risk of the orchestrator becoming a single point of failure, enhancing the overall robustness of the system.

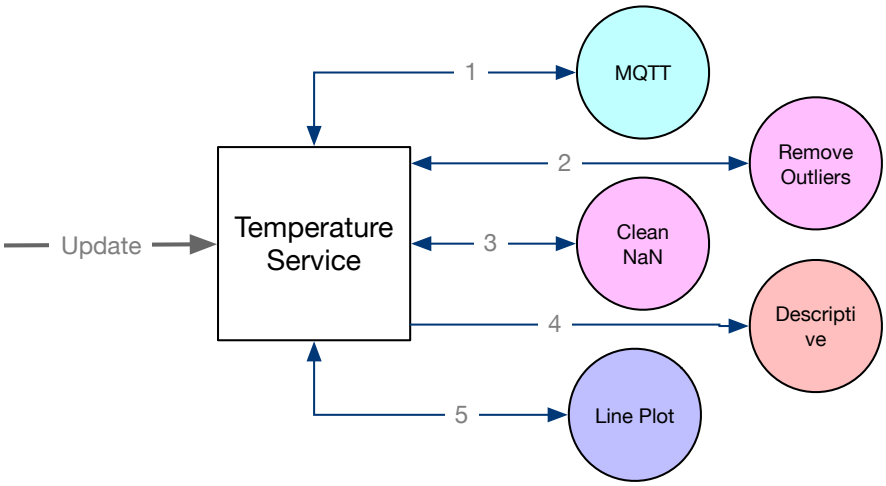

Figure 2.2: Orchestrator

2.2.2 Deployment and Containerization Technologies

Microservices deployment is facilitated by containerization, where each service, along with its dependencies, is encapsulated within a container, providing a consistent execution environment across development, testing, and production. Docker is one of the most widely used containerization tools, allowing each microservice to run independently. For managing and orchestrating containers, tools such as Kubernetes and Docker Swarm are used [10], [11].

Kubernetes provides extensive features like load balancing, self-healing, and auto-scaling, making it suitable for complex microservices applications in production environments. Kubernetes' ability to manage clusters of containers makes it ideal for applications requiring high availability and resilience [11], [14].

On the other hand, **Docker Swarm** is a simpler orchestration solution, best suited for smaller applications where complex scaling and resilience are less critical. Docker Swarm's ease of setup and native Docker integration make it attractive for straightforward applications that do not require the more advanced features of Kubernetes [10], [15].

2.3 Communication Brokers

Communication brokers are key components in microservices architectures, facilitating efficient, reliable, and scalable communication between distributed services. Various communication brokers have been extensively studied in the literature, each offering unique features suited to specific application scenarios.

Communication brokers have a rich history, tracing back to the development of Messaging-Oriented Middleware (MOM) in the 1980s. These early systems, such as IBM's MQSeries, were designed to address the need for reliable message delivery across distributed enterprise applications. However, as the requirements of distributed systems grew, early MOM systems revealed limitations in terms of flexibility and scalability.

To address these evolving needs, the messaging landscape diversified, giving rise to modern high-performance and low-latency platforms like Apache Kafka and NATS. These

newer brokers were designed for real-time applications and are now widely used in environments demanding rapid, fault-tolerant communication, such as the Large Hadron Collider monitoring systems, which utilize ActiveMQ for reliable, high-throughput messaging across various subsystems [17].

These developments have led to a deeper understanding of the role of message queues and publish-subscribe systems in distributed architectures, as they enable reliable, asynchronous communication essential for microservices.

Message queues and publish-subscribe (pub-sub) systems are fundamental components in distributed systems and microservices architecture, enabling asynchronous communication between services. These systems provide critical capabilities for flexibility, reliability, and scalability, essential in modern architectures where services operate independently but need coordinated communication.

Message Queues and Decoupling: Message queues allow producers to send messages without waiting for immediate processing by consumers. This enables each service to operate at its own pace, buffering messages to handle high loads efficiently. Research shows that message queues improve system throughput and reduce latency, making them highly suited for high-throughput microservices where responsiveness is paramount [18].

Architectural Implications and Design Principles: The integration of message queues and pub-sub systems represents a shift toward asynchronous, loosely-coupled designs that support robust service interactions. This approach enables microservices to manage diverse workloads by scaling services independently and supports Event-Driven Architecture (EDA) and command query responsibility segregation (CQRS) patterns. In EDA, brokers broadcast events in real-time, reducing latency and enhancing user responsiveness, while in CQRS, message queues manage separate read and write workloads, maintaining consistency without sacrificing speed. Together, these principles ensure that systems can adapt to both planned scaling events and unexpected traffic surges.

Reliability and Fault Tolerance: Message queues play a critical role in ensuring reliable data transmission across distributed systems. Varying delivery guarantees (e.g., "at-least-once," "exactly-once") help maintain data integrity. This reliability is essential in

applications where data loss or message ordering issues could disrupt workflows. Studies comparing message queue technologies highlight their fault tolerance and durability, which are crucial for consistency in distributed applications [18], [19].

Publish-Subscribe (Pub-Sub) Model for Real-Time Updates: The publish-subscribe model enables one-to-many communication, where a single publisher can broadcast messages to multiple subscribers, making it ideal for event-driven systems. In an e-commerce system, for instance, an "order created" event can trigger parallel actions in payment, inventory, and shipping services, all reacting to updates independently. Research in microservices architectures underscores that pub-sub systems improve scalability and real-time responsiveness by allowing multiple services to consume events simultaneously.

2.3.1 Apache Kafka

Apache Kafka is one of the most widely adopted communication brokers in modern microservices architectures. Kafka has demonstrated its effectiveness in handling large volumes of data with high throughput and durability. Chy et al. [20] and Maharjan et al. [19] highlight Kafka's role in real-time data processing and integration with machine learning frameworks and big data tools. Kafka's ability to handle message streams with low latency and high availability makes it an excellent choice for applications that require real-time data analytics.

Further studies, such as those by Ataei et al. [21], show how Kafka's publish/subscribe model has been integrated into massive IoT (MIoT) environments, improving scalability and security through blockchain-based data storage. Kafka's architecture, which includes data partitioning and replication, ensures fault tolerance and scalability, making it suitable for big data streaming applications, as noted by Le Noac'h et al. (2017) and Hiraman (2018) [22], [23].

2.3.2 ActiveMQ Artemis

ActiveMQ Artemis, the modern successor to the original ActiveMQ, has gained attention for its versatility and efficient resource utilization. Chy et al. [20] and Fu et al. [18] emphasize Artemis's ability to support various messaging protocols, making it a robust option for microservices architectures that require flexibility. Artemis is particularly suited for resource-constrained environments due to its optimized performance and balanced latency and throughput, as noted by Maharjan et al. [19].

2.3.3 RabbitMQ

RabbitMQ is known for its robust performance and support for multiple messaging protocols, which makes it a popular choice for applications requiring guaranteed message delivery, such as financial systems. RabbitMQ's routing capabilities and support for various messaging patterns have been highlighted by T and K [24] and Dobbelaere and Esmaili [25], making it an attractive option for applications prioritizing reliability and complex routing. While RabbitMQ may lag behind Kafka in terms of throughput, its lower latency metrics make it competitive for certain use cases, as noted by Maharjan et al. [19].

2.3.4 NATS

NATS, a high-performance and lightweight messaging system, is gaining popularity in microservices and IoT environments due to its low latency and simplicity. T and K [24] highlight NATS's ability to handle high-performance messaging with ease, making it suitable for cloud-native applications where low latency is critical. NATS supports multiple communication models, including publish-subscribe and request-reply, making it a flexible option for various dynamic network environments.

2.4 Challenges and Future Directions

While significant progress has been made in both ZDM and microservices architectures, several challenges remain. In ZDM, the integration of complex industrial systems with advanced data analytics and machine learning algorithms requires flexible, scalable, and technology-agnostic frameworks. The RAMI4.0 model provides a solid foundation, but existing platforms like FIWARE still need to fully align with this reference architecture, particularly in terms of modularity, scalability, and data management [26].

Similarly, in microservices architectures, the challenge lies in managing asynchronous workloads, ensuring efficient communication, and maintaining resilience in distributed systems. Batista et al. [27] suggest that managing workloads in a multi-tenant architecture requires careful design of communication brokers to ensure system scalability and fault tolerance.

As ZDM continues to evolve towards I5.0, and as microservices architectures become more widespread in industrial applications, the development of modular, scalable, and resilient platforms will be critical. A RAMI4.0 compatible microservices architecture, leveraging the power of communication brokers such as Kafka, Artemis, and RabbitMQ, will be essential in supporting the next generation of industrial CPS.

2.5 Summary

The literature underscores the significant role of service-based architectures and communication brokers in advancing ZDM and microservices in modern industrial applications. As these systems evolve, both ZDM and microservices are moving towards more flexible, dynamic, and human-centered paradigms, aligning with the principles of I4.0 and I5.0. Service-oriented frameworks like RAMI4.0 offer structured guidelines to support the integration of advanced technologies, such as DT, AI, IoT, enabling enhanced connectivity, data flow, and predictive insights across industrial processes [**Konstantinidis2023**, 1], [5].

Microservices architectures, with their modular design principles, promote scalability, resilience, and independent deployment, which are essential for managing complex, distributed industrial systems. Communication patterns such as orchestration and choreography, along with containerization platforms like Kubernetes and Docker Swarm, are critical for handling the demands of large-scale microservices deployments [10], [14], [15]. As microservices expand in industry applications, communication brokers like Apache Kafka, ActiveMQ Artemis, and RabbitMQ provide robust support for efficient, low-latency data transfer, meeting the stringent performance requirements of real-time analytics and large data streams [19]–[21].

The shift towards I5.0 further emphasizes the importance of human-centered, sustainable production systems. Emerging concepts such as XaaS facilitate flexible service deployment and data-driven decision-making, positioning ZDM and microservices as pivotal elements for future-proof, adaptive manufacturing environments [7], [11]. As the landscape of industrial manufacturing and software development advances, these architectures will continue to shape efficient, resilient, and highly scalable solutions capable of addressing the demands of modern I5.0 systems.

Chapter 3

Coordination and Communication

In distributed systems, coordination and communication between services are essential to achieving scalability, resilience, and efficient resource utilization. In particular, microservices architectures rely on these interactions to manage complex workflows, maintain data consistency, and optimize system performance under varying workloads. Communication brokers, as intermediaries that facilitate message exchange between services, form the backbone of this coordination. This chapter explores the fundamental principles of communication in distributed environments, focusing on the role of brokers in enabling reliable, asynchronous communication. By providing loosely coupled interactions and advanced features like load balancing, failover, and protocol support, communication brokers help address common challenges in distributed systems, such as network unreliability and the need for service independence.

Communication brokers form the backbone of microservices architecture by allowing independent service operation, critical to decoupling service interactions and enhancing resilience. This decoupling is fundamental for high-availability applications, where each service must remain functional and responsive, even if others fail. Brokers support various communication protocols (e.g., HTTP, WebSockets) that help integrate polyglot microservices while ensuring consistent message exchange.

Adding robust data handling features, such as queues and fault tolerance, brokers cater to high-traffic applications across industries. In e-commerce, for instance, brokers support

transaction processing and data transfer under high demand, while in IoT ecosystems, they help manage continuous data streams, ensuring that messages are not dropped even when services are temporarily offline.

3.1 Communication Brokers

Communication brokers play an essential role in microservices by enabling loosely coupled interactions among services. They provide reliable message-passing, service coordination, and a range of advanced communication features essential for distributed systems [17]. By addressing challenges such as network unreliability, strong coupling, and heterogeneous system components, modern messaging brokers ensure that services can communicate flexibly and efficiently, supporting high-throughput and low-latency environments.

3.1.1 Core Concepts in Messaging Systems

Messaging systems are designed to decouple producers and consumers, allowing them to communicate asynchronously while maintaining data consistency and reliability [17]. Key concepts include:

- Message Structure: Messages typically consist of a body containing structured data and a set of headers with metadata for routing and processing. Common data formats include JSON, XML, and serialization protocols.

- Communication Models: Most messaging systems support two primary models—queues (for point-to-point messaging) and topics (for publish/subscribe scenarios). Queues ensure that a message is delivered to a single consumer, while topics allow multiple consumers to receive the same message, supporting diverse use cases.

Expanding beyond simple queues and topics, advanced communication models combine aspects of both to handle complex consumer requirements in high-throughput scenarios. Hybrid models, which enable independent consumer scaling, optimize high-demand scenarios by balancing load across large consumer groups. In these cases, CEP (Complex Event Processing) comes into play, allowing brokers to filter, transform, and aggregate

data for real-time insights. CEP is essential in use cases such as fraud detection and telemetry monitoring, where rapid data processing is crucial.

- Protocol Support: Protocols like AMQP, STOMP, and MQTT facilitate communication across various environments, each with specific features to address different messaging needs. For instance, AMQP provides a binary protocol for interoperability, while MQTT is optimized for low-bandwidth and high-latency networks, ideal for IoT applications.

Security in messaging systems is reinforced through multiple layers, including end-to-end encryption, authentication, and authorization. Brokers such as RabbitMQ use TLS/SSL to encrypt messages in transit, while Kafka incorporates pluggable security protocols like OAuth and SASL for flexible authentication. These protocols protect sensitive data streams in sectors like finance and healthcare, ensuring compliance with industry standards like HIPAA and GDPR. Role-Based Access Control (RBAC) adds an authorization layer, securing access at a granular level and helping prevent unauthorized data flow within the microservices ecosystem.

3.1.2 Classification of Communication Brokers

Communication brokers can be categorized based on their architecture and primary functionality:

- Message Queues: These brokers (e.g., RabbitMQ, ActiveMQ) offer reliable delivery using queue mechanisms, which temporarily store messages until they are processed. They support message persistence, fault tolerance, and complex routing capabilities.

- Event Streaming Platforms: Platforms like Apache Kafka specialize in real-time, high-throughput data streaming. Unlike traditional brokers, Kafka employs a log-based architecture, where messages are retained based on time-based policies rather than immediate consumption, facilitating efficient data processing across distributed systems.

- Broker-less Systems: Although not technically brokers, libraries like ZeroMQ allow

direct communication between services without an intermediary, using advanced socket patterns to maintain loosely coupled interactions. ZeroMQ is particularly effective for high-performance applications where low latency is critical, although it lacks traditional message persistence and routing features.

3.1.3 Capabilities of Modern Messaging Systems

Modern communication brokers possess advanced features that make them essential for distributed systems, including:

- Persistence and Durability: Ensures messages survive system failures, with some brokers supporting database integration for managing message storage and retrieval. Advanced brokers like Kafka and RabbitMQ support fault tolerance through data replication and partitioning strategies. Kafka's distributed log model ensures high availability by replicating data across multiple nodes, preventing data loss in case of broker failure. Similarly, RabbitMQ's mirrored queues allow data replication across multiple nodes, supporting high availability in applications that cannot afford data loss, such as banking or healthcare. These strategies ensure continuity and reliability in fault-tolerant systems, where service interruptions could lead to significant data loss or service disruption.

- Fault Tolerance and Failover: Brokers often provide automated failover mechanisms, allowing continuous operation in case of node failures.

- Guaranteed Delivery: Delivery policies, such as at-least-once or exactly-once, ensure messages reach their intended destinations even under challenging conditions.

- Load Balancing and Clustering: Load balancing is critical in distributed systems where traffic patterns fluctuate. Brokers like NATS utilize algorithms that dynamically assign messages based on consumer load, ensuring that no single node is overwhelmed. Kafka partitions data across brokers, where consumers can access specific partitions independently, a feature particularly beneficial in high-throughput data

streaming applications like media delivery or live sports analytics. Load balancing thus contributes to brokers' scalability, making them suitable for applications that demand flexibility under load changes.

- Flexible Protocol and Language Support: Most brokers support multiple protocols and client languages, enabling integration across diverse environments.

To maintain performance, brokers integrate observability tools, such as Prometheus for monitoring metrics like message lag, throughput, and node health. Kafka, for instance, tracks partition leader status and consumer group lag, allowing proactive detection of potential bottlenecks. RabbitMQ provides tools to monitor queue health, latency, and connection status, helping administrators mitigate issues before they affect service continuity. Observability thus enhances broker reliability, ensuring stable performance even in dynamic environments where workload changes unpredictably.

3.2 Comparison and Evaluation of Communication Brokers

To select the optimal broker for ZDM, four communication brokers — Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS — were evaluated. This evaluation aimed to assess the brokers' effectiveness in real-time messaging, scalability, and fault tolerance in a microservices architecture.

3.2.1 Broker Setup and Configuration

To ensure consistent performance comparisons, brokers were tested in a uniform environment using Docker containers to replicate production-like network conditions. Metrics such as CPU usage, memory consumption, and network latency were checked, providing a comprehensive view of each broker's resource efficiency. By standardizing the environment, the tests accurately reflect each broker's scalability, making the results applicable

to a wide range of real-world use cases, from light-load web applications to high-frequency trading systems The following communication brokers were used:

- **Apache Kafka**: Kafka is well-known for its ability to handle high-throughput, fault-tolerant messaging scenarios. It is widely used in streaming applications and real-time data processing due to its robust architecture [20].
- **ActiveMQ Artemis**: This broker is a successor to the original ActiveMQ, designed for enterprise-level applications. It supports a variety of messaging protocols and patterns, making it flexible for different use cases [18].
- **RabbitMQ**: RabbitMQ is known for its ease of integration with various platforms and its ability to handle complex routing tasks using the Advanced Message Queuing Protocol (AMQP). It is frequently used in scenarios requiring reliable message delivery [20].
- **NATS**: NATS is optimized for simplicity and high performance in lightweight applications. It supports fast, scalable communication in cloud-native environments and IoT ecosystems [28].

3.2.2 Testing and Performance Evaluation

Each broker was tested under various communication scenarios, including Java-to-Java, Java-to-Python, Python-to-Java, and Python-to-Python communications. The brokers were evaluated across multiple metrics:

- **Latency**: The time taken for a message to be delivered from the producer to the consumer. This metric is critical in real-time data processing environments. NATS showed consistently low latency across scenarios, particularly excelling in Python-to-Python and Java-to-Python communications. For low message volumes (100 messages), NATS achieved an impressive 0.02 seconds in Python-to-Python, making it ideal for scenarios requiring minimal delay, such as IoT data collection or live feeds.

- **Throughput**: The number of messages successfully delivered per second. This is a key indicator of how well the broker handles large volumes of messages. Kafka emerged as a strong choice for high-throughput applications, maintaining reliable message delivery rates even at 100,000 messages. In Python-to-Java communication, Kafka's performance remained stable with an average delay of 42 seconds, which, while not the fastest, supports applications prioritizing consistent delivery, such as sequential data streams [22], [29].

- **Scalability**: The ability of each broker to handle increasing message volumes while maintaining performance. NATS demonstrated exceptional scalability with minimal latency increase across volumes, reaching only 21.85 seconds at the highest volume (100,000 messages) in Python-to-Python communication. RabbitMQ also displayed strong scalability in Java-to-Java messaging, reaching a latency of 14 seconds at very high loads, suitable for applications needing reliable cross-language communication at high message volumes.

- **Reliability**: The brokers were tested for their ability to recover from network disruptions and maintain consistent message delivery [21]. Kafka's partitioned architecture allowed it to maintain high message integrity, with minimal message loss even under high-volume conditions. NATS demonstrated high reliability by quickly recovering from disruptions with minimal message loss, particularly effective in lightweight, cloud-native setups. RabbitMQ experienced some message loss under extreme conditions, especially in Python-to-Java, while ActiveMQ Artemis saw the highest message loss, indicating limitations in diverse, high-load environments.

The following results represented in heatmaps outline the broker performance across different communication scenarios, including Python-to-Python, Python-to-Java, Java-to-Python, and Java-to-Java. The darker colors indicate slower sending and receiving, while white shows faster exchange of messages.

The tests were performed for 100, 1000, 10000, and 100000 messages, measuring both

the producer and the consumer. For 100 messages (Figure 3.1), the best results were obtained with the combination of NATS and Python as both consumer and producer.

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	0.1	0.11	0.13	0.04
NATS	0.02	0.02	0.09	0.09
Artemis	0.13	0.6	2.1	1.28
Kafka	0.05	0.45	0.5	0.5

(a) Producer Performance

Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
0.08	0.1	0.07	0.04
0.03	0.04	0.03	0.04
5.2	1.52	1.8	1.21
0.03	0.1	0.1	0.12

(b) Consumer Performance

Figure 3.1: 100 messages

For 1000 messages (Figure 3.2), the best results were obtained with the combination of NATS and Python as both consumer and producer, just like in the previous case.

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	0.85	0.9	0.25	0.3
NATS	0.17	0.19	0.22	0.22
Artemis	1.2	1.2	11.87	11.61
Kafka	0.28	0.47	0.99	1

(a) Producer Performance

Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
0.85	0.85	0.54	0.29
0.21	0.22	0.22	0.25
7.8	3.21	10.5	11.24
0.29	0.91	0.69	0.81

(b) Consumer Performance

Figure 3.2: 1000 messages

For 10,000 messages (Figure 3.3), the best results were obtained with the combination of RabbitMQ and Java as producer and Python as consumer. NATS continued to show low latency, recording 2.03 seconds in Python-to-Python, reflecting its suitability for high-throughput applications such as IoT.

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	8	8.01	1.44	1.6
NATS	2.03	2.04	1.19	1.25
Artemis	11.49	11.51	98.77	98.24
Kafka	5.02	4.39	6.22	6.1

(a) Producer Performance

Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
8	8	5.26	1.67
2.1	2.06	2.31	2.2
14.37	29.6	92.8	97.85
3.65	6.06	5.92	5.76

(b) Consumer Performance

Figure 3.3: 10,000 messages

Finally, for 100,000 messages (Figure 3.4), the best results were obtained with NATS and Java as both consumer and producer, close to the Python implementation. RabbitMQ

maintained its performance at 14 seconds for Java-to-Java, suitable for high-load Java applications.

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	77.38	78.22	66.66	14.66
NATS	21.85	20.03	23.71	19.94
Artemis	116.06	124.97	949.13	949.66
Kafka	42.71	42.73	57.03	55.39

(a) Producer Performance

Broker	Pt-Pt	Pt-Jv	Jv-Pt	Jv-Jv
RabbitMQ	77.37	78.55	55.54	14.69
NATS	21.9	20.04	25	20.87
Artemis	122.56	292	944.69	949.26
Kafka	41.16	51.79	55.22	54.9

(b) Consumer Performance

Figure 3.4: 100,000 messages

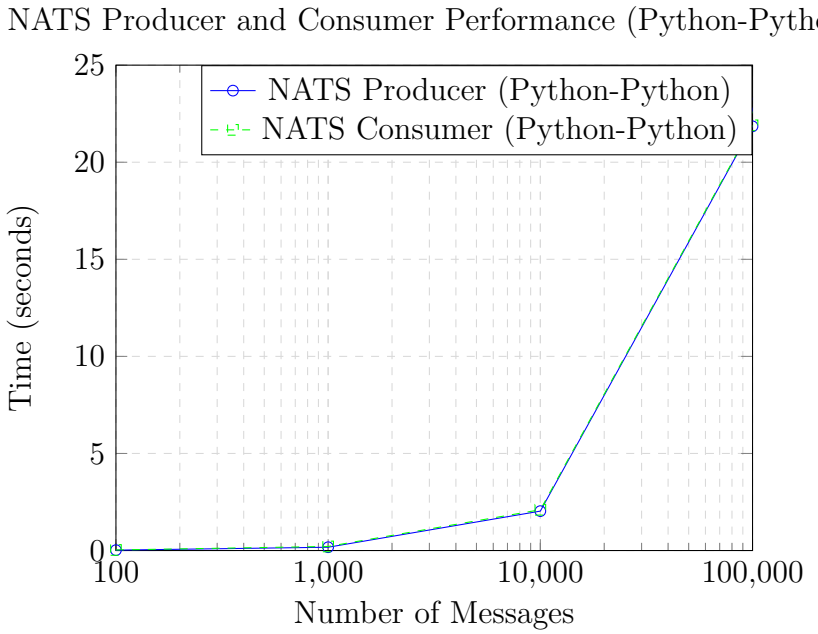

Figure 3.5: NATS Producer and Consumer Performance (Python-Python)

The comparative analysis demonstrates NATS as an optimal choice for high-performance, low-latency applications due to its lightweight, cloud-native architecture, which aligns well with ZDM requirements. In our ZDM architecture, which will involve Python-to-Python communication, NATS's exceptionally low latency and scalability make it ideal for rapid message exchange in real-time data processing. Unlike other brokers with complex configurations and higher resource demands, NATS operates with a minimal footprint, supporting fast deployment and efficient resource use. This efficiency enables quicker communication

cycles without compromising scalability or reliability.

Additionally, NATS's simplicity in setup and management, combined with its built-in fault tolerance, makes it an attractive solution for organizations needing straightforward yet powerful messaging capabilities. Its "at-most-once" and "at-least-once" delivery options ensure reliable data transmission in microservices-based environments where precise control over message delivery guarantees is necessary. NATS's support for both publish/-subscribe and point-to-point messaging models, combined with its resilience under heavy workloads, highlights its robustness in handling complex data streams that are essential for ZDM and other real-time processing systems.

The broker's streamlined protocol support, especially HTTP and WebSocket integration, simplifies interoperability across diverse microservices and polyglot architectures. This flexibility reduces the overhead associated with bridging language and protocol gaps between services, enhancing efficiency and minimizing latency even under scaled-up, cross-language communication scenarios.

In summary, NATS's efficient resource usage, low-latency performance, and flexibility make it a compelling choice for ZDM's microservices-based approach, where both responsiveness and resilience are essential to maintain continuous, zero-defect operations. This reasoning supports the decision to favor NATS as the backbone communication broker for this architecture.

3.3 Use Cases and Example Implementations

Messaging has become a foundation for large-scale distributed systems, with notable implementations in high-stakes environments. CERN's Large Hadron Collider (LHC) Monitoring Systems employ ActiveMQ to manage data collection and monitoring across its particle accelerator network, ensuring reliable data flow while preventing system overload [17]. In high-energy physics data acquisition, the ATLAS experiment at CERN utilizes a messaging framework to distribute operational alarms and metadata across geographically dispersed sites. Additionally, the Worldwide LHC Computing Grid employs

a STOMP-based messaging system to support over 50,000 clients globally, providing high infrastructure reliability.

3.4 Summary

In this chapter, we explored the foundational role of communication brokers in supporting microservices architecture, enabling reliable and efficient message-passing across distributed systems. We began by introducing communication brokers and tracing their evolution from early MOM systems to modern high-performance platforms, such as Apache Kafka and NATS, that address the complexities of real-time data streaming and fault tolerance in distributed environments. Core concepts, including message structure, communication models, and protocol support, were examined to illustrate how brokers facilitate asynchronous, loosely coupled service interactions.

We categorized brokers into message queues, event streaming platforms, and brokerless systems, each offering unique capabilities suited to specific distributed system needs. Furthermore, we reviewed essential capabilities of modern brokers—such as persistence, fault tolerance, and load balancing—highlighting how these features contribute to system resilience, reliability, and scalability in microservices.

To determine the optimal broker for ZDM, we conducted an in-depth comparison of four brokers: Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS. Each was tested across multiple metrics, including latency, throughput, scalability, and reliability under different scenarios, such as Java-to-Java and cross-language communications. The evaluation revealed distinct strengths and limitations for each broker, with Kafka excelling in throughput for high-volume scenarios and NATS demonstrating excellent scalability for lightweight, cloud-native applications. RabbitMQ provided robust integration capabilities, while ActiveMQ Artemis was effective for enterprise-grade routing and protocol flexibility, albeit with higher latency in high-volume settings.

Finally, real-world implementations showcased the versatility of communication brokers, with examples like CERN's LHC Monitoring Systems and the Worldwide LHC

Computing Grid underscoring brokers' critical role in enabling scalable, fault-tolerant data handling across complex, distributed infrastructures.

In summary, communication brokers are indispensable in advancing microservices and distributed systems, providing the messaging backbone for a wide array of applications. Their evolving capabilities make them vital for addressing the growing demands of scalability, resilience, and performance in modern data-driven environments.

Chapter 4

Microservices for ZDM Architecture

Achieving zero defects in manufacturing demands a robust and flexible system to capture and process vast data streams in real-time, allowing proactive adjustments to ensure quality. The ZDM approach, underpinned by I4.0 principles, requires the integration of CPS, DT, and advanced analytics to create a resilient ecosystem where defects are minimized at every stage of production. This microservices-based framework promotes modularity, flexibility, and scalability across manufacturing domains, aligning with RAMI4.0 standards for interoperability and data-driven insights, ensuring adaptability to changing production demands and customer requirements [30], [31].

This chapter introduces the ZDM architecture's microservices system, detailing its data pipeline, modular approach, and deployment within an automotive assembly line to demonstrate real-world capabilities in data acquisition, rule-based analytics, and real-time monitoring for effective defect reduction.

4.1 Architecture and Pipeline of Microservices

The proposed ZDM architecture decomposes traditional monolithic structures into specialized microservices, each responsible for distinct stages of the data pipeline: **Data Collection, Processing, Analytics, and Visualization**. This modular structure promotes adaptability, enabling easy integration of new technologies within manufacturing

environments and supporting continuous deployment [6].

Key components in the ZDM architecture include:

- **Data Collection Microservices**: These microservices gather raw data from sensors via AAS, serving as digital twins that communicate asset-specific data. Lightweight protocols like MQTT facilitate low-latency, real-time data ingestion [32].
- **Data Processing Microservices**: These apply schema-based validation, error correction, and normalization using tools like Apache Avro to ensure data consistency [33].
- **Data Analytics Microservices**: This stage involves analytical microservices programmed to apply specific rule-based models for quality assessment, such as Nelson rules. These models enable real-time detection of quality anomalies, identifying trends and deviations that indicate potential defects.
- **Visualization Microservices**: These services display results on dashboards, enabling operators to monitor real-time metrics, detect deviations, and take prompt corrective action.

Each microservice is designed for specific tasks like data validation, error correction, and continuous monitoring, ensuring independence and fault tolerance across the system. The architecture's modularity allows the deployment of each service independently, aligning with RAMI4.0 standards for interoperability and predictive analytics [32], [34].

4.1.1 Introduction to the Architecture

The ZDM architecture is structured to address complex requirements for defect-free manufacturing by decomposing the traditional monolithic system into specialized microservices, each with a distinct role in managing data flow. This modular architecture allows each microservice to perform targeted functions independently, enabling the system to be both adaptable and fault-tolerant.

The core data pipeline is divided into four main stages:

- **Data Collection** – Acquiring raw data from sensors and other input sources.
- **Data Processing** – Pre-processing and validating data, ensuring it adheres to specified schemas.
- **Data Analytics** – Analyzing processed data to detect patterns, anomalies, or defects.
- **Data Visualization** – Presenting analyzed data in real-time dashboards for easy monitoring.

In breaking down these stages, the architecture achieves modularity, scalability, and flexibility, as each stage can be optimized or scaled independently based on demand. Additionally, this structure allows for continuous deployment, meaning that updates to one microservice (e.g., adding new analytical algorithms) do not disrupt the entire system, a critical feature for high-availability production environments.

Description of Each Pipeline Component

Data Collection Microservices Data Collection Microservices gather real-time data from manufacturing sensors and monitoring devices along the assembly line. In an automotive factory, for instance, sensors measure factors such as motor temperature, torque, and alignment accuracy of components in real time. These microservices use protocols such as MQTT, which is lightweight and optimized for resource-constrained devices, enabling frequent and efficient data transmission. Types of Data Collected includes measurements of part alignment, critical for ensuring components are accurately and securely fitted.

Data Processing Microservices Data Processing Microservices perform essential preprocessing functions, transforming raw sensor data into a format suitable for analytics. These services validate data structures against Avro schemas, perform normalization, and correct any detected errors.

- **Schema-Based Validation**: Each data point is validated according to a predefined schema (e.g., Avro) that ensures consistency across services. For example, torque measurements have specific value ranges and data types that are checked automatically.

- **Error Correction**: When sensors transmit inconsistent or incorrect values—such as readings much higher than the limit a pre-set correction protocol to adjust values or discard unreliable data is used.

- **Detailed Example**: In the automotive assembly line, a processing microservice could normalize alignment measurements taken from various sensor models to ensure data uniformity. These measurements are structured into a unified Avro format, with deviations corrected based on historical data trends for each component, enhancing the accuracy of subsequent analytics.

Data Analytics Microservices Data Analytics Microservices apply rule-based models, machine learning algorithms, and statistical methods to detect patterns or anomalies in the processed data. These models aim to predict defects or identify trends that signal potential issues, allowing preventive action to be taken.

- **Types of Analyses**:

  - *Rule-Based Models (e.g., Nelson Rules)*: Rule-based models flag sudden deviations in motor temperature or torque values, which may indicate excessive wear or incorrect assembly techniques.

  - *Machine Learning Algorithms*: Clustering or anomaly detection algorithms can highlight unusual patterns, such as a consistent misalignment trend across multiple chassis, which might indicate a systemic calibration issue.

  - *Statistical Process Control*: Control charts track parameters over time, identifying trends that could signify issues that appear over time.

- **Example of Defect Prediction**: Suppose an analytics microservice identifies a pattern of increasing torque in specific fasteners on the engine assembly line. By detecting this pattern early, the microservice flags the trend as a potential defect risk due to overtightening, notifying operators to adjust the torque settings before significant rework is needed.

Visualization Microservices Visualization Microservices convert processed data and analytics into human-readable formats, usually presented on real-time dashboards. These dashboards enable operators to monitor essential KPIs, analyze trends, and respond to alerts quickly.

- **Dashboard Components**:
  - *Real-Time Alerts*: Visual indicators (e.g., color-coded alerts) highlight any critical issues, such as torque exceeding specified limits on fasteners.
  - *Trend Graphs and Analytics*: Real-time and historical trends in assembly precision enable operators to spot emerging issues proactively.
  - *Interactive Filters and Drill-Down Options*: Operators can focus on specific assembly processes or filter by date range to review data from recent shifts, providing insights for targeted inspections.
- **Example Dashboard Setup**: In the automotive assembly plant, the dashboard displays metrics such as bolt torque and part alignment accuracy across all assembly stations. Operators can isolate metrics for specific processes, like engine mounting or chassis assembly, and view predictive trends based on past data, helping them take preventive measures well before defects manifest.

4.1.2 Benefits of a Modular Microservices-Based Architecture

- **Flexibility in Component Scaling**: Each microservice can be independently scaled according to its demand. For example, Data Collection services handling

high-frequency sensor data can be scaled horizontally by adding more instances, while Data Analytics services may only require additional resources during peak production hours.

- **Fault Tolerance and System Resilience**: The architecture supports fault tolerance by enabling individual microservices to fail or restart without affecting the entire system. If a data collection microservice fails due to a network issue, other services (e.g., Data Processing) continue to function, ensuring minimal disruption.

- **Continuous Deployment and Rapid Updates**: Each microservice can be updated or modified independently, reducing deployment complexity. For instance, a new algorithm in the Data Analytics service can be deployed without downtime for other services, ensuring the system remains up-to-date with minimal disruption.

- **Enhanced Observability and Troubleshooting**: With each microservice performing a specific function, it is easier to monitor, troubleshoot, and optimize each part of the system. Logging, metrics, and alerts are tailored for each microservice, providing better insights into system performance and allowing rapid diagnosis of issues.

4.1.3 Practical Example: End-to-End Data Flow in a ZDM Setup

To illustrate the complete data flow through this pipeline, consider the following scenario in an automotive assembly line using the ZDM architecture:

1. **Data Collection**: Sensors on the assembly line track motor part alignment. These readings are collected by Data Collection Microservices, which send the data to the MQTT broker for real-time relay.

2. **Data Processing**: The Data Processing Microservices pull data from the broker,

validate it against Avro schemas, and correct any minor inconsistencies. For example, if a part alignment reading falls outside the expected range, an error correction protocol adjusts the reading based on historical averages.

3. **Data Analytics**: The analytics microservice then applies Nelson Rules to identify anomalies. If it detects a trend indicating that measurements are increasing or decreasing, suggesting potential problems in assembly, it flags this trend for immediate review.

4. **Visualization**: An alert appears on the dashboard in the control room, showing the change in measurements with a "critical" tag. Operators can drill down into the data, observing real-time graphs of the measurements, and initiate corrective action on affected equipment.

This end-to-end example underscores the ZDM microservices architecture's ability to detect, process, and act on data in real-time, demonstrating its role in maintaining zero defects in manufacturing.

4.1.4 Schema Management and Transformation: The Role of Data Schemas in Microservices

In distributed microservices-based systems like those used in Zero Defect Manufacturing (ZDM), **data schemas** form the backbone of structured data exchange. A data schema defines the format, structure, and constraints of data, providing a blueprint for data integrity, interoperability, and validation across different services. In a complex system like ZDM, schemas are crucial for ensuring that data flows accurately and efficiently from sensors through to analytics platforms and visualization tools.

Importance of Data Schemas in Distributed Architectures

Data schemas are particularly essential in distributed microservices for several reasons:

- **Consistency and Accuracy**: By enforcing a standardized data format, schemas ensure that data is uniformly interpreted across different services. This is critical in ZDM, where quality metrics and sensor data must be consistent to detect and prevent defects effectively.
- **Validation and Data Integrity**: Predefined data structures allow for real-time data validation, which prevents erroneous or malformed data from entering the system. This reduces downstream errors in analytics and visualization, improving the overall robustness of the ZDM process.
- **Interoperability**: In microservices, where diverse components may use different languages and data protocols, schemas provide a shared "language" that ensures smooth communication between services.
- **Performance Optimization**: Efficient data exchange is critical in high-throughput systems. Adherence to standardized formats reduces the need for costly transformations and enables faster processing.

Types of Data Schemas and Their Applications

Data schemas can be defined using various formats, each suitable for different types of data and system requirements. Key schema formats include relational database schemas, document schemas, Avro schemas, XML Schema Definition (XSD), and Protocol Buffers. Here, we focus particularly on Avro schemas due to their significance in OpenZDM.

Avro Schema Apache Avro is a data serialization framework that uses a compact binary format and is optimized for high-throughput data systems like Apache Kafka, which is commonly employed in streaming and real-time applications. Avro's schema-based serialization format allows for efficient data encoding and facilitates schema evolution, a feature particularly important in systems where data requirements change over time. This makes Avro ideal for ZDM environments, where continuous updates to manufacturing processes often necessitate schema changes[35].

Avro schemas consist of data fields, their types, and metadata describing the data. In OpenZDM, each sensor reading or process log entry is serialized using Avro, enabling consistent, lightweight data transport from production lines to analytics systems without excessive bandwidth consumption.

Example Avro Schema for Power Consumption Sensor:

Listing 4.1: Example Avro Schema for Power Consumption Sensor

```
{
  "type": "record",
  "name": "PowerSensorReading",
  "namespace": "com.zdm.sensors",
  "fields": [
    { "name": "sensorId", "type": "string" },
    { "name": "timestamp", "type": "long" },
    { "name": "voltage", "type": "float" },
    { "name": "current", "type": "float" },
    { "name": "power", "type": "float" }
  ]
}
```

The Avro schema above defines a basic structure for power consumption data, including fields for `sensorId`, `timestamp`, `voltage`, `current`, and `power`. This structure provides ZDM with a standardized format for power data, enabling real-time data ingestion, validation, and transformation without complex processing.

The use of Avro also supports **schema evolution**, which is crucial in ZDM environments[36]. In Avro, schemas can evolve without breaking compatibility, thanks to a schema registry (e.g., Apicurio) that manages schema versions. This capability allows OpenZDM to add or modify fields in sensor data without disrupting the entire system, providing both flexibility and stability.

Comparative Overview of Schema Types

While Avro is a primary choice in OpenZDM, other schema formats are also relevant in various contexts. Table 4.1 summarizes these formats and their applications.

Schema Type	**Description**	**Typical Use Cases**
Relational Schema	Structured table-based schema with fixed columns	CRM, ERP
Document Schema (JSON, BSON)	Flexible, hierarchical schema for dynamic data	E-commerce, Content Management
Avro Schema	Compact, binary serialization for high throughput	IoT, Real-time analytics
XML Schema (XSD)	Strict hierarchical schema with strong validation	Finance, Healthcare
Protocol Buffers (ProtoBuf)	Efficient, binary schema for RPC	High-performance APIs

Table 4.1: Overview of Common Schema Types

Example Scenarios Demonstrating Schema Use

Example 1: Customer Profile Data in E-commerce (JSON Schema) An e-commerce platform might store customer profiles with fields like customer ID, purchase history, and browsing preferences. Here's a simplified JSON schema to validate these profiles[37]:

Listing 4.2: JSON Schema for Customer Profile

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Customer Profile",
  "type": "object",
  "properties": {
    "customerId": { "type": "string" },
    "name": { "type": "string" },
    "email": { "type": "string", "format": "email" },
    "purchases": {
      "type": "array",
      "items": {
        "type": "object",
```

```
        "properties": {
          "productId": { "type": "string" },
          "purchaseDate": { "type": "string", "format": "date" },
          "amount": { "type": "number" }
        },
        "required": ["productId", "purchaseDate", "amount"]
      }
    }
  },
  "required": ["customerId", "name", "email"]
}
```

This schema ensures data consistency across customer profiles, validating required fields like email format and purchase records. JSON schemas like this one provide e-commerce platforms with a flexible structure that adapts easily to changes in customer data requirements.

Example 2: Real-Time Sensor Data in Smart Grid (Avro Schema) In a smart grid system, data from sensors measuring power consumption can be serialized with Avro for compact and fast transmission:

Listing 4.3: Example Avro Schema for Smart Grid Sensor

```
{
  "type": "record",
  "name": "SmartGridReading",
  "namespace": "com.grid.sensors",
  "fields": [
    { "name": "sensorId", "type": "string" },
    { "name": "timestamp", "type": "long" },
    { "name": "voltage", "type": "float" },
```

```
    { "name": "current", "type": "float" },
    { "name": "status", "type": "string" }
  ]
}
```

This Avro schema allows the smart grid system to efficiently transmit real-time sensor data, supporting schema evolution for new sensor types and data attributes, and ensuring high compatibility between old and new data formats.

Flexible Schema Management Options

In addition to schema formats, schema management strategies also vary based on application needs:

- **Schema-on-Read vs. Schema-on-Write**: Schema-on-read applies schema validation at retrieval, useful in flexible data environments. Schema-on-write enforces validation on ingestion, which ensures that only clean, validated data enters the system[38].

- **Schema Registries**: Tools like Apicurio or Confluent Schema Registry store schema versions and enforce compatibility rules, making them invaluable for data platforms like Kafka[39].

- **Decentralized Schema Management**: In decentralized setups, each service defines its schema independently, promoting flexibility but potentially leading to inconsistencies if governance policies are not enforced.

Conclusion

Data schemas provide the foundation for structured data exchange in ZDM architectures, ensuring data consistency, validation, and interoperability across microservices. Avro schemas, in particular, offer efficient serialization and robust schema evolution, making them ideally suited to the dynamic data requirements of ZDM systems.

4.1.5 Data Transformation and Communication Flow

In the Data Collection stage, sensors measure the alignment and fitment of vehicle parts, formatting data into AAS-compliant structures. The data entry point, known as the Non-Destructive Inspection (NDI), serves as the primary access point for real-time data from the factory's AAS. The NDI transforms incoming data into an Avro schema format, ensuring uniformity, readability, and validation across microservices. By mapping raw data to predefined fields, this transformation reduces manual intervention and enhances data consistency. Avro's compact serialization supports efficient processing, while Apicurio Registry manages schema evolution to ensure backward and forward compatibility as the ZDM process evolves [20].

The NATS messaging system manages asynchronous communication across services, supporting high-throughput, low-latency transmission essential for quality issue responses. Figure 4.1 illustrates the overall architecture flow, from data capture at the inspection stations to final visualization and reporting.

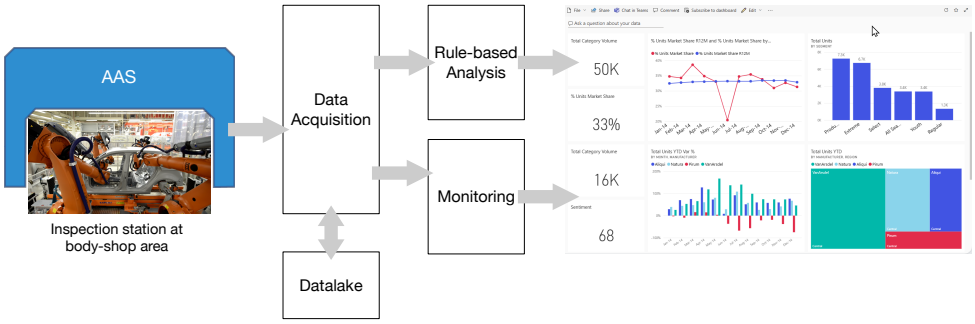

Figure 4.1: Overview of the experimental implementation pipeline

4.2 Pipeline Implementation and Industrial Use Cases

The ZDM pipeline implementation is structured to facilitate continuous monitoring and quality assurance across different manufacturing stages. Within an automotive assembly line, each pipeline stage serves a specific function that collectively maintains product quality, reduces downtime, and optimizes resource allocation. These stages include Data Lake

Integration, Rule-Based Analysis, and Real-Time Visualization, each critical to sustaining defect-free operations.

The following subsections expand on these core use cases, demonstrating how ZDM microservices work in a complex, high-stakes production environment.

4.2.1 Data lake Integration for Long-Term Analysis

Data lake stores vast amounts of raw and processed data generated across long periods of time. In the automotive sector, a data lake serves as a historical archive for parameters such as alignment specifications. Data Collection Microservices continually send sensor data to MongoDB, where it is stored as structured records in Avro format, preserving schema consistency and enabling rapid retrieval for analysis.

- **Purpose**: The data lake provides an extensive history of quality metrics, supporting retrospective analysis and aiding the development of predictive maintenance schedules. This is crucial for identifying recurring patterns or anomalies, such as slight deviations in measurements over weeks, which may indicate a problem in assembly tools.

- **Integration Example**: During each shift, the alignment data from chassis mounting stations is logged. This data, stored in a data lake, allows analysts to correlate alignment deviations with specific machine states, operational conditions, or staff shifts, creating a repository for ongoing quality improvement.

- **Operational Benefits**: Storing data long-term enables root-cause analysis, allowing quality control teams to detect patterns like misalignments correlating with tool degradation or process changes. Over time, this historical data informs adjustments in preventive maintenance schedules.

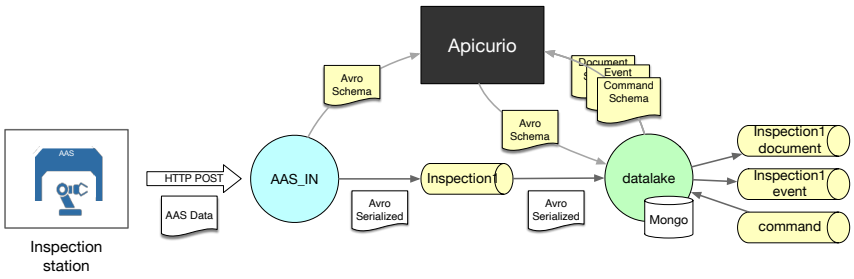

Figure 4.2: Datalake integration pipeline

4.2.2 Rule-based Analysis for Real-Time Quality Control

Rule-based analysis applies predefined models, such as Nelson rules, to identify patterns that might indicate potential defects. Analytics Microservices analyze sensor data in real time, applying statistical rules to detect anomalies. For instance, a sudden increase in a measurement reading on a specific assembly line station could indicate that equipment calibration is needed.

- **Purpose**: Real-time rule-based analysis is essential for immediate identification of variations that fall outside acceptable ranges. This approach helps prevent defects by enabling timely corrective action on the assembly line, such as adjusting machine settings or inspecting tools before they cause a failure.

- **Industrial Application**: For example, specific measurement data from a certain point is continuously monitored. When the readings exceed a pre-defined range, the rule-based system issues an alert, prompting an operator to inspect the equipment for possible malfunctions or recalibration needs.

- **Feedback Mechanism**: Alerts triggered by analytics microservices are logged and stored, providing a dataset for future optimization of rule parameters. Engineers may adjust tolerance levels based on these historical alerts, gradually refining the rule-based models to better reflect real-world operating conditions.

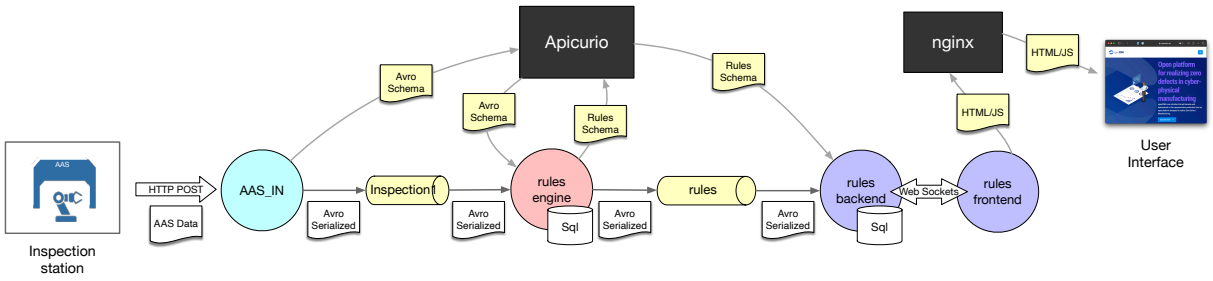

Figure 4.3: Rule-based analysis pipeline for quality control

4.2.3 Real-Time Data Visualization and Operator Dashboard

The real-time data visualization component translates complex sensor data into actionable insights displayed on operator dashboards. Visualization Microservices aggregate data from the data lake and analytics modules, presenting it in a user-friendly format. Operators can see live metrics, such as current torque levels, temperature stability, and alignment precision, which help them make informed adjustments on the fly.

- **Purpose**: Dashboards provide an immediate overview of critical metrics, helping operators stay responsive to any variations. This visual feedback loop supports proactive management of quality across all production stages, reducing the likelihood of defects progressing through the line undetected.

- **Dashboard Features**:

  - *Real-Time Alerts*: The dashboard highlights any deviations in critical metrics, like sudden fluctuations in alignment readings, with color-coded warnings to signal immediate attention.

  - *Performance Trends*: Trends in alignment are visualized, allowing operators to anticipate equipment recalibrations or shifts in process efficiency.

  - *Historical Comparison*: Operators can view recent data alongside historical baselines, enabling them to identify emerging trends that might signal impending defects.

- **Benefits of Real-Time Visualization**: Dashboards equipped with drill-down capabilities allow operators to focus on specific sections or equipment, providing a view that aids rapid problem diagnosis. This real-time oversight helps maintain optimal performance and reduces potential downtime.

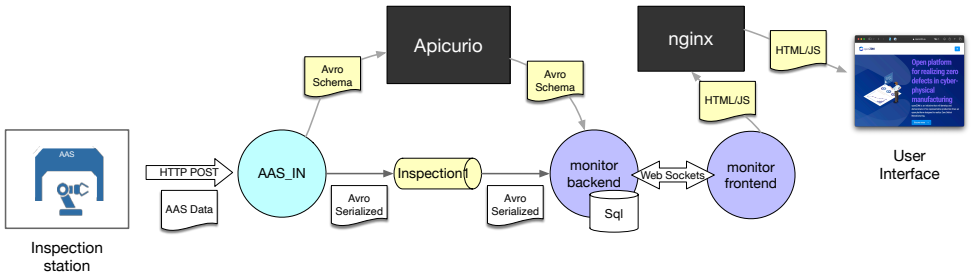

Figure 4.4: Real-time data visualization pipeline

By implementing these three core use cases within the ZDM architecture, the automotive assembly line ensures a responsive, data-driven production environment. Data Lake Integration, Rule-Based Analysis, and Real-Time Visualization collectively support the ongoing pursuit of defect-free manufacturing by providing layers of data validation, analysis, and monitoring.

4.3 Infrastructure and Deployment Considerations

The ZDM framework's infrastructure includes NATS as the primary message broker, Apicurio for schema registry, and Docker/Kubernetes for container management and deployment. Each microservice maintains independence by storing only the data necessary for its functionality, enhancing modularity and fault tolerance.

Figure 4.5 displays the services managing the decoupled microservices, such as the message broker, schema registry, deployment registry, and API gateway. The NATS channels provide high-throughput communication between modules, while the Apicurio schema registry ensures dynamic adaptation to various data structures.

Performance tests under varying loads (10, 100, 1000, and 10,000 messages) yielded response times of 7, 12, 64, and 635 seconds, respectively. This scalability demonstrates

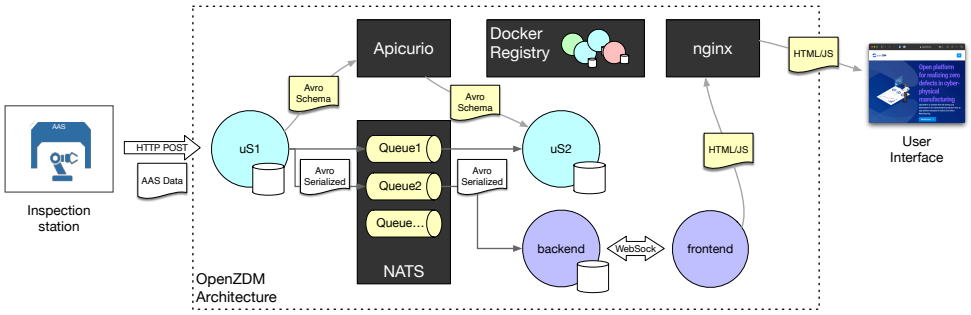

Figure 4.5: Overview of the microservices management services

that the architecture handles both standard and peak workloads effectively, providing continuous operation with minimal latency.

4.3.1 Role of NATS in High-Performance Messaging for ZDM

In the ZDM architecture, NATS offers high throughput and low latency, which is essential for supporting real-time communication in quality management processes. NATS's lightweight protocol minimizes network overhead, and its clustered setup ensures balanced message distribution across nodes, increasing fault tolerance and preventing bottlenecks under varying loads.

Clustered Configuration for Fault Tolerance and Load Distribution NATS operates in a clustered configuration, enhancing fault tolerance and load distribution by enabling each microservice to route messages through the nearest available node. This reduces latency and ensures continuous message flow, especially crucial for applications where delays could lead to product quality issues. Additionally, NATS's failover mechanism allows for automatic re-routing of messages in case of node failure, maintaining system resilience and minimizing downtime.

NATS and Avro for Efficient Data Serialization NATS integrates seamlessly with Avro for efficient, schema-based data serialization. Avro's compact binary format minimizes the size of each message, optimizing NATS for high-volume data processing with minimal bandwidth consumption. This pairing supports the dynamic schema evolution

required in ZDM, where production parameters are regularly updated to reflect new quality standards. This setup enables NATS to handle real-time data transmissions while maintaining schema consistency across microservices.

4.3.2 Containerized Deployment and Scalability with Docker and Kubernetes

The ZDM architecture leverages containerized deployment, using Docker for lightweight packaging and Kubernetes for orchestration, scaling, and fault isolation. This approach facilitates seamless updates and continuous deployment, allowing each microservice to operate independently while maintaining communication integrity within the NATS messaging framework.

Blue-Green Deployment for Continuous Operations The ZDM environment utilizes a blue-green deployment strategy to reduce downtime during updates. This deployment model creates a staging environment (blue) for testing new features and configurations while the production environment (green) continues operating. Upon successful testing, the blue environment replaces green, enabling rapid updates without affecting production. This is critical in ZDM, where production disruptions can lead to significant operational costs[40].

Real-Time Metrics for System Health and Performance Metrics such as consumer lag, message processing delay, and resource utilization are tracked in real-time, providing insights into bottlenecks or underperforming nodes. For example, monitoring NATS message queue health helps identify delays or network congestion, allowing for immediate intervention. This proactive approach ensures that production maintains its zero-defect objective by continuously optimizing resource allocation and system responsiveness.

In summary, the infrastructure and deployment strategies implemented within the

ZDM framework enhance its resilience, scalability, and real-time responsiveness. By leveraging NATS's low-latency messaging, Kubernetes's container orchestration, and advanced observability tools, the ZDM architecture effectively addresses the high-performance demands of modern manufacturing.

4.4 Summary

This chapter presented an in-depth exploration of a microservices-based architecture for ZDM, designed to enhance flexibility, resilience, and scalability in modern manufacturing environments. By integrating foundational I4.0 principles, including CPS, DT, and RAMI4.0 standards, the proposed architecture creates a responsive ecosystem where data-driven insights drive proactive quality management and defect reduction.

The ZDM architecture's data pipeline, composed of Data Collection, Processing, Analytics, and Visualization microservices, was detailed to demonstrate each stage's role in maintaining data integrity and enabling real-time decision-making. Each microservice independently manages specific tasks—such as schema validation, error correction, rule-based analytics, and interactive data visualization—ensuring modularity and fault tolerance. The architecture leverages NATS as a high-performance message broker, ensuring low-latency, scalable communication, while Apicurio manages schema evolution, allowing the system to adapt seamlessly to production changes.

Real-world implementation examples from an automotive assembly line illustrated how data flows through the ZDM pipeline, ensuring continuous monitoring, data validation, and anomaly detection. Core use cases, including data lake integration for long-term trend analysis, rule-based quality assessments using Nelson rules, and real-time dashboards for operator monitoring, underscored the architecture's capabilities in delivering timely insights for proactive quality interventions.

The chapter also discussed the infrastructure and deployment considerations critical to maintaining a high-performing ZDM system. The framework's use of Kubernetes for

container orchestration and the blue-green deployment strategy enables continuous operations with minimal downtime.

In conclusion, this microservices-based architecture for ZDM fosters an environment where quality is continuously monitored, and defects are anticipated rather than merely detected. With technologies such as NATS for scalable messaging and Apicurio for schema management, the ZDM framework provides a robust foundation for future expansions into predictive and prescriptive maintenance, laying the groundwork for a fully adaptive manufacturing ecosystem. This architecture represents a significant step forward in achieving zero-defect production, combining data-driven insights with agile, resilient infrastructure in a way that aligns with evolving manufacturing demands and quality standards.

Chapter 5

Conclusions

The integration of ZDM within I4.0 frameworks, combined with microservices architecture, offers a transformative approach to achieving defect-free, resilient, and adaptable manufacturing systems. This thesis explored how microservices—coupled with advanced communication brokers like Apache Kafka, ActiveMQ Artemis, RabbitMQ, and NATS—can enhance real-time data exchange, proactive defect management, and system scalability across production lines, thereby supporting the rigorous requirements of ZDM.

The research began by addressing limitations inherent in monolithic architectures, which struggle to meet the demands of dynamic, data-intensive environments. Microservices architecture emerged as an agile and modular alternative, enabling independent, loosely coupled services that facilitate fault tolerance, flexible scaling, and high availability. These attributes are essential to sustaining ZDM in manufacturing environments where reliability, responsiveness, and efficiency are paramount. By decomposing production systems into distinct microservices for data collection, processing, analytics, and visualization, this architecture enables a robust ZDM framework that is highly adaptive to fluctuations in manufacturing conditions.

Through rigorous performance evaluation, the thesis assessed communication brokers—Kafka, RabbitMQ, ActiveMQ Artemis, and NATS—highlighting their unique strengths

for various ZDM applications. Kafka demonstrated resilience in high-throughput scenarios, making it ideal for streaming large data volumes with consistency. RabbitMQ excelled in handling complex message routing and guaranteed message delivery, while NATS emerged as particularly efficient for low-latency, lightweight communication. ActiveMQ Artemis, while versatile, showed greater latency under high load, indicating its suitability for specific enterprise applications rather than scenarios requiring rapid response. These insights are critical for practitioners selecting messaging frameworks tailored to their specific manufacturing environments.

A major contribution of this thesis is the proposed microservices-based architecture for ZDM, validated through a case study in automotive manufacturing. Implementing this architecture in a real-world production line highlighted its capability to manage complex workflows, detect and prevent defects in real-time, and support high-frequency data monitoring. By leveraging modular microservices, each responsible for specific functions such as data validation, rule-based analysis, and interactive visualization, this system proved to be highly effective in maintaining production quality under fluctuating workloads. The use of a communication broker like NATS reinforced the architecture's real-time responsiveness, ensuring efficient data flow and minimal latency.

The findings of this thesis have several significant implications for both industrial practitioners and software architecture design:

1. **Microservices Architecture as a Catalyst for ZDM**: By enabling modular, scalable, and resilient systems, microservices have shown their efficacy in managing the demands of modern manufacturing environments. These environments benefit from increased adaptability, fault tolerance, and flexibility, which are crucial for defect prevention and rapid responsiveness to production changes.

2. **Critical Role of Communication Brokers**: The evaluation of communication brokers highlighted their essential function in facilitating scalable, asynchronous communication. As industries adopt microservices architectures, selecting an appropriate communication broker will be fundamental to achieving the desired levels

of throughput, latency, and delivery guarantees necessary for effective ZDM.

3. **Practical Considerations for Broker Selection**: The thesis provided actionable insights for selecting communication brokers based on specific needs—whether for high-throughput, complex routing, or low-latency requirements. These findings assist practitioners in choosing messaging solutions aligned with the unique requirements of their production systems, ultimately enhancing overall system performance and reliability.

While the research demonstrated the advantages of microservices and communication brokers, several challenges remain, presenting opportunities for future research:

- **Managing Orchestration and Choreography Complexity**: As microservices architectures expand in scale, managing inter-service communication becomes increasingly complex, particularly in large distributed systems. Future research could explore more advanced orchestration and choreography techniques, potentially incorporating artificial intelligence and machine learning to optimize service coordination and adapt workflows dynamically.
- **Expanding Real-Time Predictive Analytics**: Although rule-based analytics proved effective for immediate defect detection, there is substantial potential in integrating predictive maintenance models using advanced machine learning algorithms. Future studies could delve into AI-driven predictive analytics to further enhance ZDM's proactive capabilities, leveraging historical data to predict defects and implement preventive measures before issues arise.
- **Ensuring Data Security and Privacy**: As microservices handle larger data volumes, safeguarding data security and privacy remains a critical concern, particularly when sensitive manufacturing data is involved. Developing robust security frameworks that maintain performance without compromising data integrity and privacy will be essential, especially as ZDM systems are integrated into broader networks and supply chains.

- **Scaling Across Heterogeneous Environments**: While this thesis demonstrated scalability within a controlled industrial environment, future research should explore implementation across diverse and distributed manufacturing networks. Such environments introduce additional variables such as latency, data governance, and regulatory compliance, which will require new strategies for maintaining system integrity and data consistency.

In conclusion, this thesis has underscored the transformative potential of microservices architecture and communication brokers in advancing ZDM. The proposed architecture facilitates real-time data exchange, supports continuous scalability, and promotes fault tolerance, offering a foundation for future innovations in industrial manufacturing systems. As I5.0 evolves, with its focus on sustainable, human-centered production, the principles, insights, and technologies presented in this research will play a pivotal role in shaping resilient, adaptive, and defect-free manufacturing systems. This work contributes not only to the ongoing discourse on modern industrial architectures but also to the development of next-generation ZDM frameworks that prioritize quality, scalability, and responsiveness in an increasingly complex manufacturing landscape.

Bibliography

[1] D. Powell, M. C. Magnanini, M. Colledani, and O. Myklebust, "Advancing zero defect manufacturing: A state-of-the-art perspective and future research directions," *Computers in Industry*, vol. 136, p. 103 596, 2022, ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2021.103596. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361521002037.

[2] F. Konstantinidis, V. Balaska, S. Symeonidis, *et al.*, "Achieving Zero Defected Products in Diary 4.0 using Digital Twin and Machine Vision," English, in *ACM International Conference Proceeding Series*, 2023, pp. 528–534, ISBN: 9798400700699. DOI: 10.1145/3594806.3596554.

[3] M. Magnanini, M. Colledani, and D. Caputo, "Reference architecture for the industrial implementation of zero-defect manufacturing strategies," English, in *Procedia CIRP*, ISSN: 2212-8271, vol. 93, 2020, pp. 646–651. DOI: 10.1016/j.procir.2020.05.154.

[4] P. Martinez, M. Al-Hussein, and R. Ahmad, "A cyber-physical system approach to zero-defect manufacturing in light-gauge steel frame assemblies," English, in *Procedia Computer Science*, ISSN: 1877-0509, vol. 200, 2022, pp. 924–933. DOI: 10.1016/j.procs.2022.01.290.

[5] V. Medici, M. Martarelli, N. Paone, *et al.*, "Integration of Non-Destructive Inspection (NDI) systems for Zero-Defect Manufacturing in the Industry 4.0 era," English, in *IEEE International Workshop on Metrology for Industry 4.0 and IoT*, 2023,

pp. 439–444, ISBN: 9798350396577. DOI: 10.1109/MetroInd4.0IoT57462.2023.10180016.

[6] E. Solvsberg, C. Oien, S. Dransfeld, R. Eleftheriadis, and O. Myklebust, "Analysis-oriented structure for runtime data in Industry 4.0 asset administration shells," English, in *Procedia Manufacturing*, ISSN: 2351-9789, vol. 51, 2020, pp. 1106–1110. DOI: 10.1016/j.promfg.2020.10.155.

[7] S. Zeb, A. Mahmood, S. Khowaja, *et al.*, "Towards defining industry 5.0 vision with intelligent and softwarized wireless network architectures and services: A survey," English, *Journal of Network and Computer Applications*, vol. 223, 2024, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2023.103796.

[8] D. Lindsay, S. S. Gill, D. Smirnova, and P. Garraghan, "The evolution of distributed computing systems: From fundamental to new frontiers," *Computing*, vol. 103, no. 8, pp. 1859–1878, 2021, ISSN: 1436-5057. DOI: 10.1007/s00607-020-00900-y. [Online]. Available: https://doi.org/10.1007/s00607-020-00900-y.

[9] P. K. Donta, I. Murturi, V. Casamayor Pujol, B. Sedlak, and S. Dustdar, "Exploring the potential of distributed computing continuum systems," *Computers*, vol. 12, no. 10, 2023, ISSN: 2073-431X. DOI: 10.3390/computers12100198. [Online]. Available: https://www.mdpi.com/2073-431X/12/10/198.

[10] "Soa vs microservices - difference between architectural styles," *AWS Architecture Blog*, n.d. Accessed: 2024-10-13. [Online]. Available: https://aws.amazon.com/architecture/soa-vs-microservices/.

[11] "What is service-oriented architecture (soa)?" Accessed: 2024-10-13, IBM Cloud Education. (n.d.), [Online]. Available: https://www.ibm.com/cloud/learn/soa.

[12] S. Sengupta, "Service-oriented architecture vs microservices architecture: Comparing soa to msa," *BMC Software Blog*, 2021, Accessed: 2024-10-13. [Online]. Available: https://www.bmc.com/blogs/soa-vs-microservices/.

[13] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Springer, 2003, Accessed: 2024-10-13. [Online]. Available: https://link.springer.com/book/10.1007/978-3-662-05208-3.

[14] "Microservices vs. service-oriented architecture." Accessed: 2024-10-13, Baeldung on Computer Science. (n.d.), [Online]. Available: https://www.baeldung.com/cs/microservices-vs-soa.

[15] "The benefits of microservices choreography vs orchestration." Accessed: 2024-10-13, Solace. (n.d.), [Online]. Available: https://solace.com/blog/microservices-choreography-vs-orchestration/.

[16] C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in microservice architecture," *Executive Director - Solutions Architect Media and Entertainment domain*, 2023.

[17] L. Magnoni, "Modern messaging for distributed sytems," *Journal of Physics: Conference Series*, vol. 608, no. 1, p. 012 038, Apr. 2015. DOI: 10.1088/1742-6596/608/1/012038. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/608/1/012038.

[18] G. Fu, Y. Zhang, and G. Yu, "A fair comparison of message queuing systems," *IEEE Access*, vol. 9, pp. 421–432, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3046503.

[19] R. Maharjan, M. Chy, M. Arju, and T. Cerny, "Benchmarking message queues," *Telecom*, vol. 4, no. 2, pp. 298–312, 2023, ISSN: 2673-4001. DOI: 10.3390/telecom4020018.

[20] M. Chy, M. Arju, S. Tella, and T. Cerny, "Comparative evaluation of java virtual machine-based message queue services: A study on kafka, artemis, pulsar, and RocketMQ," *Electronics (Switzerland)*, vol. 12, no. 23, 2023, ISSN: 2079-9292. DOI: 10.3390/electronics12234792.

[21] M. Ataei, A. Eghmazi, A. Shakerian, R. Landry Jr., and G. Chevrette, "Publish/-subscribe method for real-time data processing in massive IoT leveraging blockchain for secured storage," *Sensors*, vol. 23, no. 24, 2023, ISSN: 1424-8220. DOI: 10.3390/s23249692.

[22] P. Le Noac'h, A. Costan, and L. Bougé, "A performance evaluation of apache kafka in support of big data streaming applications," in *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, 2017, pp. 4803–4806.

[23] B. R. Hiraman, "A study of apache kafka in big data stream processing," in *2018 International Conference on Information, Communication, Engineering and Technology (ICICET)*, IEEE, 2018, pp. 1–6.

[24] S. T and S. N. K, *A study on modern messaging systems- kafka, RabbitMQ and NATS streaming*, Dec. 8, 2019. DOI: 10.48550/arXiv.1912.03715. arXiv: 1912.03715[cs]. [Online]. Available: http://arxiv.org/abs/1912.03715 (visited on 04/03/2024).

[25] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, 2017, pp. 227–238.

[26] G. Angione, C. Cristalli, J. Barbosa, and P. Leitao, "Integration challenges for the deployment of a multi-stage zero-defect manufacturing architecture," English, in *IEEE International Conference on Industrial Informatics (INDIN)*, ISSN: 1935-4576, Helsinki-Espoo, Finland, 2019, pp. 1615–1620, ISBN: 978-1-72812-927-3. DOI: 10.1109/INDIN41052.2019.8972259.

[27] C. Batista, F. Morais, E. Cavalcante, T. Batista, B. Proença, and W. B. Rodrigues Cavalcante, "Managing asynchronous workloads in a multi-tenant microservice enterprise environment," *Software: Practice and Experience*, vol. 54, no. 2, pp. 334–359, Feb. 2024, ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.3278. [Online].

Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.3278 (visited on 01/16/2024).

[28] *Nats.io – cloud native, open source, high-performance messaging*, https://nats.io/, Accessed: 2024-05-13.

[29] H. Wu, Z. Shang, and K. Wolter, "A reactive batching strategy of apache kafka for reliable stream processing in real-time," *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 294–305, 2020.

[30] M. Attaran and B. Celik, "Digital twin: Benefits, use cases, challenges, and opportunities," *Decision Analytics Journal*, vol. 6, 2023, ISSN: 2772-6622. DOI: 10.1016/j.dajour.2023.100165.

[31] DIN Deutsches Institut für Normung e. V., *DIN SPEC 91345: Referenzarchitekturmodell Industrie 4.0 (RAMI4.0)*, https://www.din.de/de/forschung-und-innovation/themen/industrie4-0/din-veroeffentlicht-din-spec-zu-rami4-0-158570, Accessed: 11 March 2024, 2016.

[32] C. Wagner, J. Grothoff, U. Epple, *et al.*, "The role of the industry 4.0 asset administration shell and the digital twin during the life cycle of a plant," presented at the IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, ISSN: 1946-0740, 2017, pp. 1–8, ISBN: 978-1-5090-6505-9. DOI: 10.1109/ETFA.2017.8247583.

[33] N. Dragoni, S. Giallorenzo, A. L. Lafuente, *et al.*, "Microservices: Yesterday, Today, and Tomorrow," en, in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds., Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67424-7 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. [Online]. Available: http://link.springer.com/10.1007/978-3-319-67425-4_12 (visited on 03/03/2024).

[34] J. Frysak, C. Kaar, and C. Stary, "Benefits and pitfalls applying RAMI4.0," in *IEEE Industrial Cyber-Physical Systems (ICPS)*, St. Petersburg, May 2018, pp. 32–37,

ISBN: 978-1-5386-6531-2. DOI: 10.1109/ICPHYS.2018.8387633. [Online]. Available: https://ieeexplore.ieee.org/document/8387633/ (visited on 02/21/2024).

[35] A. S. Foundation, "Apache avro documentation," *Apache Avro*, 2023, https://avro.apache.org/docs/current/. [Online]. Available: https://avro.apache.org/docs/current/.

[36] R. Le Noac'h, P. Bonnet, and S. Bouchenak, "A performance evaluation of distributed concurrency control," in *Proceedings of the 2017 Symposium on Cloud Computing*, ACM, 2017, pp. 68–82. DOI: 10.1145/3127479.3128617.

[37] A. Smith and R. Kumar, "Efficient json data processing in e-commerce systems," in *2021 IEEE International Conference on Big Data*, IEEE, 2021, pp. 1618–1623. DOI: 10.1109/BigData52589.2021.9671765.

[38] N. Patil and H. Shah, *Practical Data Schema Management in Real-Time Systems*, 1st. Sebastopol, CA: O'Reilly Media, 2020.

[39] C. Inc., *Schema registry documentation*, 2023. [Online]. Available: https://docs.confluent.io/platform/current/schema-registry/index.html.

[40] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010, pp. 211–215, ISBN: 978-0321601919.