Article

A Scalable Framework for Sensor Data Ingestion and Real-Time Processing in Cloud Manufacturing

Massimo Pacella [1,*], Antonio Papa [2], Gabriele Papadia [1] and Emiliano Fedeli [3]

[1] Department of Engineering for Innovation, University of Salento, 73100 Lecce, Italy; gabriele.papadia@unisalento.it
[2] Department of Science and Information Technology, Pegaso University, 80121 Napoli, Italy; antonio.papa@unipegaso.it
[3] Laser Romae S.r.l., 00144 Roma, Italy; emiliano.fedeli@laserromae.it
* Correspondence: massimo.pacella@unisalento.it

Abstract: Cloud Manufacturing enables the integration of geographically distributed manufacturing resources through advanced Cloud Computing and IoT technologies. This paradigm promotes the development of scalable and adaptable production systems. However, existing frameworks face challenges related to scalability, resource orchestration, and data security, particularly in rapidly evolving decentralized manufacturing settings. This study presents a novel nine-layer architecture designed specifically to address these issues. Central to this framework is the use of Apache Kafka for robust, high-throughput data ingestion, and Apache Spark Streaming to enhance real-time data processing. This framework is underpinned by a microservice-based architecture that ensures a high scalability and reduced latency. Experimental validation using sensor data from the UCI Machine Learning Repository demonstrated substantial improvements in processing efficiency and throughput compared with conventional frameworks. Key components, such as RabbitMQ, contribute to low-latency performance, whereas Kafka ensures data durability and supports real-time application. Additionally, the in-memory data processing of Spark Streaming enables rapid and dynamic data analysis, yielding actionable insights. The experimental results highlight the potential of the framework to enhance operational efficiency, resource utilization, and data security, offering a resilient solution suited to the demands of modern industrial applications. This study underscores the contribution of the framework to advancing Cloud Manufacturing by providing detailed insights into its performance, scalability, and applicability to contemporary manufacturing ecosystems.

Keywords: Cloud Manufacturing; innovative framework; layered architecture; system scalability; resource management; high-demand workloads

Academic Editors: Sheng Du, Zixin Huang, Li Jin and Xiongbo Wan

Received: 18 November 2024
Revised: 27 December 2024
Accepted: 2 January 2025
Published: 4 January 2025

Citation: Pacella, M.; Papa, A.; Papadia, G.; Fedeli, E. A Scalable Framework for Sensor Data Ingestion and Real-Time Processing in Cloud Manufacturing. *Algorithms* **2025**, *18*, 22. https://doi.org/10.3390/a18010022

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Cloud Manufacturing (CM) represents a significant advancement in the manufacturing sector, combining the power of Cloud Computing (CC) and the Internet of Things (IoT) to create a seamless, interconnected production ecosystem. This innovative approach allows the integration of geographically dispersed manufacturing resources, including equipment, materials, and expertise, into a unified platform. Thus, the CM enables manufacturers to access and utilize resources on demand, similar to how CC services are consumed [1,2]. This paradigm shift not only enhances operational efficiency, but also promotes flexibility and scalability in manufacturing processes.

As illustrated in Figure 1, the pay-per-use model inherent in CM offers manufacturers numerous advantages. This allows companies to access advanced manufacturing capabilities without the need for significant upfront investment in equipment and infrastructure. This model optimizes resource allocation by ensuring that manufacturing assets are utilized more efficiently across different users and projects. Furthermore, it enables smaller enterprises to compete in a more level-playing field with larger corporations by providing access to state-of-the-art manufacturing technologies and expertise. Consequently, CM is poised to revolutionize the manufacturing industry by fostering innovation, reducing costs, and improving overall productivity.

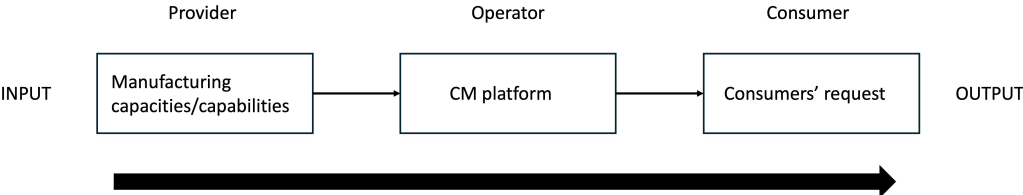

Figure 1. Overview of CM platform architecture. Within this framework, providers produce data regarding their manufacturing capabilities, which are subsequently managed by an operator, whereas consumers receive services customized to their specific requirements.

The integration of CC and IoT technologies facilitates the development of scalable and adaptable production systems, enabling manufacturers to meet the dynamic demands of contemporary manufacturing effectively. Despite their potential, the current CM frameworks encounter significant challenges related to scalability, resource management, and data security, particularly in decentralized and rapidly evolving environments [3]. Addressing these limitations is essential for enhancing the operational efficiency of CM systems.

Recent advancements have highlighted the critical importance of resource optimization and scheduling algorithms for improving overall system efficiency [4]. Advanced scheduling methods, for instance, contribute to optimized resource allocation, reduced production lead times, and improved system performance. However, existing solutions often lack the robustness required to manage the complexities of modern manufacturing, such as volatile demand patterns and integration of diverse heterogeneous resources [5].

To address these challenges, this study proposes an innovative nine-layer framework for CM that incorporates Apache Kafka for high-throughput data ingestion and Apache Spark Streaming for real-time data processing, thereby enhancing operational efficiency and resilience [6]. Additionally, microservice architecture underpins the framework, promoting scalability and reducing latency to facilitate rapid adaptation to market changes, particularly within the manufacturing industry sector [7].

Experimental validation of the framework using sensor data from the University of California-Irvine (UCI) Machine Learning Repository [8] revealed significant improvements in processing time and throughput compared with traditional systems. Notably, RabbitMQ low-latency capabilities, combined with Kafka data durability, optimize the performance for real-time applications. Furthermore, Spark Streaming in-memory processing enables timely insights, making it highly suitable for dynamic data analysis in complex manufacturing scenarios [9]. Therefore, the proposed nine-layer framework enhances operational efficiency, resource optimization, and data security and establishes a resilient solution tailored to the demands of modern industrial applications. This study highlights the potential of this framework to advance CM by presenting practical insights into its performance, scalability, and applicability to contemporary manufacturing ecosystems.

The remainder of this paper is organized as follows. Section 2 presents a review of the recent state-of-the-art CM frameworks and related advanced technologies. Section 3 introduces a novel nine-layer framework designed to enhance CM systems with a modu-

lar service-oriented architecture that promotes efficient, secure, and scalable operations, aligned with the recent advancements in CM framework design. Section 4 outlines the evaluation metrics applied to assess the performance of the proposed framework along with a description of the case study utilized in this research. Section 5 presents a numerical analysis of the performance comparison between the two technologies in data ingestion systems. The numerical results are discussed in Section 6, and Section 7 presents the conclusions of the study and directions for future research. An Appendix A section is also provided, which presents a series of algorithms corresponding to the key layers of the proposed framework.

2. State-Of-The-Art Review

CM has transformed the manufacturing sector by shifting from traditional models to flexible service-centric frameworks. This paradigm enables the integration and efficient utilization of distributed manufacturing resources within a cloud ecosystem [10]. CM excels in resource sharing, coordination, and scalability, which are critical for multi-location operations involving diverse stakeholders. However, its implementation poses challenges, such as handling vast data volumes, real-time analytics, security, and collaboration across dynamic manufacturing environments. Effective CM systems require robust data ingestion, processing, and application layers to meet these demands efficiently. Technologies for CM frameworks include Apache Kafka and RabbitMQ for data ingestion, Spark Streaming for real-time analytics, MongoDB for data management, Kubernetes for orchestration, Flask-SocketIO for real-time communication, and microservice architectures for application scalability. We review these technologies as follows:

Apache Kafka — is a distributed event streaming platform. It enables high-throughput fault-tolerant data pipelines and is a pivotal system for managing large-scale real-time data streams in distributed CM environments. The log-based architecture of the Kafka platform ensures durability and order preservation, which makes it ideal for sequence-sensitive applications [11]. In the framework proposed by [12], Kafka was used to capture and manage IoT data streams, thereby significantly enhancing the scalability and resilience of the CM systems. This design leverages the strengths of Kafka in sequence-sensitive large-scale data management, enabling real-time decision making and operational efficiency within a cloud-based manufacturing context.

RabbitMQ — offers low-latency messaging, making it effective for real-time notifications and anomaly detection in CM. However, its scalability and durability are limited compared to those of Kafka, which handles continuous, large-scale data streams. The strength of RabbitMQ lies in its latency-sensitive applications, which complement those of Kafka in the hybrid systems.

Spark Streaming — supports real-time analytics in CM, enabling predictive insights and anomaly detection through continuous data processing [9,13]. Unlike Hadoop, which relies on batch processing, Spark's in-memory architecture delivers low-latency performance essential for quality control and operational intelligence [14].

MongoDB — is a NoSQL database, which is able to managing unstructured and semi-structured data such as machine logs and sensor outputs. Its scalability and JSON-based architecture make it ideal for diverse CM datasets [15]. MongoDB ensures reliable data storage and retrieval across the distributed cloud resources.

Kubernetes orchestrates containerized applications, facilitating deployment, scaling, and maintenance of CM services [16]. Its load-balancing and resource-allocation capabilities ensure resilience and scalability under variable operational demands [17], thereby enabling a high performance in dynamic CM environments.

Microservice architectures are preferred in CM owing to their modular design, which enables independent deployment, scaling, and maintenance [18]. They enhance scalability and flexibility compared to monolithic systems, which face challenges in managing large-scale environments. Microservices support evolving workloads and ensure rapid deployment and update [19].

Flask-SocketIO extends the capabilities of the Flask Web framework by enabling real-time bidirectional communication between clients and servers. This is particularly beneficial for CM systems that require instantaneous updates such as monitoring production metrics or facilitating interactive dashboards. By leveraging WebSocket protocols, Flask-SocketIO ensures low-latency communication and enhances the responsiveness in dynamic manufacturing environments. Its integration into CM frameworks supports applications such as real-time notifications, collaborative tools, and anomaly detection, further improving operational efficiency [20,21].

Security Issues

Security is a primary concern in CM, where diverse stakeholders operate within a shared cloud infrastructure. Robust security measures are particularly critical for Additive Manufacturing (AM), which is highly data-intensive and relies on intellectual property for three-dimensional (3D) models and design specifications. Unauthorized access, data breaches, and intellectual property theft can severely disrupt AM workflow and CM systems. Advanced access control mechanisms such as Role-Based Access Control (RBAC) are pivotal for addressing these threats. Flask, a lightweight web framework, is frequently used to implement RBAC in order to ensure secure resource access based on user roles. Li et al. [22] demonstrate how Flask supports CM frameworks by restricting access to authorized personnel like operators, managers, or suppliers.

Security and access control layers in CM frameworks should incorporate robust cybersecurity protocols tailored to the AM. These include encryption technologies for safeguarding data during transmission and storage, intrusion detection systems (IDSs) to identify unauthorized activities, and zero-trust architectures to enforce strict verification-based access. Ref. [23] provides a framework for assessing cybersecurity risks in multidisciplinary settings, which is vital for CM systems due to their complex structures.

Safeguarding intellectual property and 3D models in AM is critical. Zafar et al. [24] emphasized that breaches can compromise the entire manufacturing process and advocate solutions, such as digital watermarking and blockchain, for traceability and data integrity. AI-driven real-time threat intelligence further strengthens security by proactively detecting vulnerabilities and mitigating risk. For example, machine learning models can analyze network traffic to flag anomalies that are indicative of cyberattacks. Sriram and Rambabu [25] highlighted the importance of such AI-based approaches in Industry 4.0, where interconnected systems and IoT devices increase the exposure to cybersecurity risks. Their findings also emphasized fostering cybersecurity awareness among personnel to reduce human error-induced breaches. Finally, adopting multifactor authentication (MFA) and advanced cryptographic algorithms can significantly enhance security. Combining these measures with IDS and incident response frameworks establishes comprehensive defenses. Cains et al. [23] underscored the value of contextualized risk assessments for tai-

loring cybersecurity strategies and enabling CM systems to balance flexibility with rigorous security requirements.

3. The Proposed Framework

Following the recent advancements in CM framework design [26,27], this section presents a novel nine-layer framework designed to enhance CM systems with a modular service-oriented architecture that promotes efficient, secure, and scalable operations. The framework depicted in Figure 2 is organized into nine layers: (1) data collection, (2) communication, (3) data ingestion, (4) data processing and analytics, (5) application, (6) storage, (7) elastic computing and orchestration, (8) security and access control, and (9) user interface and visualization [28].

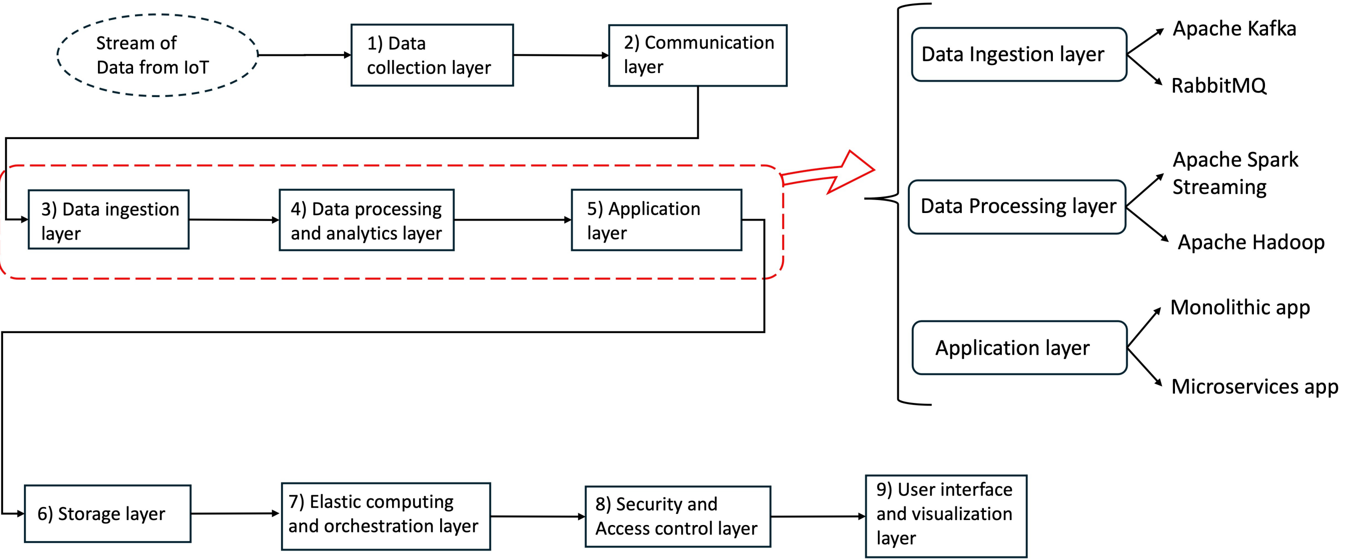

Figure 2. Schematic representation of the proposed framework. The dashed red rectangle highlights the focus of this study.

The data collection layer gathers raw data from various sources, which are then transmitted through the communication layer. The data ingestion, processing, analytics, and application layers form the core of the system, wherein data are transformed into actionable insights. The storage layer ensures data persistence, while the elastic computing and orchestration layers manage resource allocation. The security and access control layers safeguard the entire system, whereas the user interface and visualization layers present processed information in an accessible format.

In existing architectures, tools such as Apache Kafka, Spark Streaming, and microservices have been widely employed to address challenges in CM environments. For instance, frameworks leveraging RabbitMQ or Apache Kafka for data ingestion have demonstrated varying successes in managing high-throughput data streams. However, RabbitMQ often struggles with scalability in distributed systems, whereas the Kafka subscribe architecture offers exceptional scalability and fault tolerance. Similarly, Hadoop-based systems have historically dominated data processing, but have been increasingly replaced by Spark Streaming because of their low-latency, real-time processing capabilities. Microservices have gradually supplanted monolithic architectures, owing to their flexibility in independently scaling, updating, and managing discrete services [29,30].

The proposed nine-layer framework extends the existing solutions by integrating and optimizing these tools to achieve a higher degree of responsiveness, scalability, and modularity. Notably, it introduces a seamless integration of these technologies, paired with innovations, such as enhanced elastic computing mechanisms, advanced orchestration

strategies, and comprehensive data privacy and access control policies. This enables the framework to satisfy the critical demands of modern CM systems, such as real-time adaptability, robust security, and user-friendly operations. Specifically, the framework was designed to process the sensor readings generated by devices operating in the AM field, such as 3D printers. These devices produce large datasets that capture critical parameters such as temperature and relative humidity during the material extrusion process when printing polymer filaments. The choice of this dataset highlights the applicability of the framework for optimizing the production workflow in AM.

The data ingestion layer utilizes Apache Kafka, an open-source streaming platform that supports high-throughput real-time data streaming with exceptional scalability and fault tolerance. The publish-subscribe architecture of Kafka enables efficient queuing and seamless data flow to downstream services, making it ideal for CM environments in which IoT devices continuously generate high volumes of data. The experimental findings in Section 5 demonstrate that Kafka significantly outperforms RabbitMQ in handling high-velocity data streams, thereby achieving lower latency and improved scalability within distributed CM systems [31].

In the data processing and analytics layer, Spark Streaming provides real-time analytics capabilities and supports the integration of Machine Learning models for adaptive data-driven decision making. Unlike batch-oriented systems such as Hadoop, Spark Streaming offers low-latency processing, which is essential in dynamic CM environments where immediate insights optimize production workflows. As demonstrated in Section 5, Spark Streaming outperforms Hadoop in managing continuous data flows, yielding substantial reductions in processing time and enhancing operational efficiency [32].

The microservice application layer enables deployment of the framework as a modular service-oriented architecture. This configuration supports the independent scaling, updating, and management of discrete services for specific CM functions. Unlike monolithic architectures, which often encounter scalability limitations and tight inter-component coupling, the microservice approach offers enhanced flexibility and resilience. Each service operates independently, allowing rapid updates without affecting overall system stability [33,34]. Validation results (Section 5) demonstrate that the microservice architecture substantially improves system responsiveness and adaptability in high-demand manufacturing environments [35].

Furthermore, the framework emphasizes the necessity for robust data privacy and user authentication policies within the security and access control layers. These policies include encryption of sensitive data, fine-grained access control, and multifactor authentication mechanisms. Such measures ensure the confidentiality and integrity of the data and safeguard them against unauthorized access and cyber threats, which are critical for trust and compliance in IoT-driven manufacturing systems [36].

By integrating Kafka, Spark Streaming, and a microservice architecture, along with innovations in elastic computing and advanced security protocols, the proposed framework comprehensively addresses the challenges of scalability, responsiveness, and modularity inherent in IoT-driven manufacturing [37]. The experimental validations detailed in the following sections confirm the potential of the framework to significantly advance CM by enabling real-time, adaptable, and secure operations.

4. Materials and Methods

This section outlines the evaluation metrics applied to assess the performance of the proposed framework along with a description of the case study utilized in this article.

4.1. Evaluation Metrics

To assess the effectiveness of the framework, a set of established metrics targeting the system efficiency, scalability, and responsiveness was selected [38]. Based on previous research on real-time data processing, CC, and distributed systems, these metrics provide a comprehensive view of a system's operational capacity and adaptability [1,39,40].

- Throughput measures the number of messages or requests processed per second. Throughput is a key metric for industrial applications such as CM, where a higher throughput indicates an enhanced capacity for handling large-scale data streams.
- Latency defines the delay between data generation and consumption or the response time to an API request. Low latency is critical for real-time applications, where delays can impact decision making and control, especially in time-sensitive manufacturing environments [41].
- Scalability assesses the system ability to sustain performance under increasing workloads by testing with additional message producers, consumers, or concurrent users. This metric is vital for cloud-based architectures, where sensor density or data generation rates can vary significantly over time.
- Resource Utilization: Tracks CPU and memory usage to gauge efficiency. Effective resource utilization ensures high performance and cost efficiency because high utilization may indicate bottlenecks, whereas low utilization and high throughput suggest an optimized system design.

Using these metrics [42], the accuracy and performance of the framework are evaluated using established methodologies in the CC field.

4.2. Case Study

The core layers of the proposed framework, namely the data ingestion, data processing and analytics, and application architecture were validated using real-world sensor data from the "Istanbul Stock Exchange Real-Time Data for Temperature and Humidity Sensors". This dataset was sourced from the "Appliances EnergyPrediction Data Set" on the UCI Machine Learning Repository [43]. It includes temperature and humidity readings from various rooms in a low-energy residential building, with a sample of 100 consecutive readings, as shown in Figure 3.

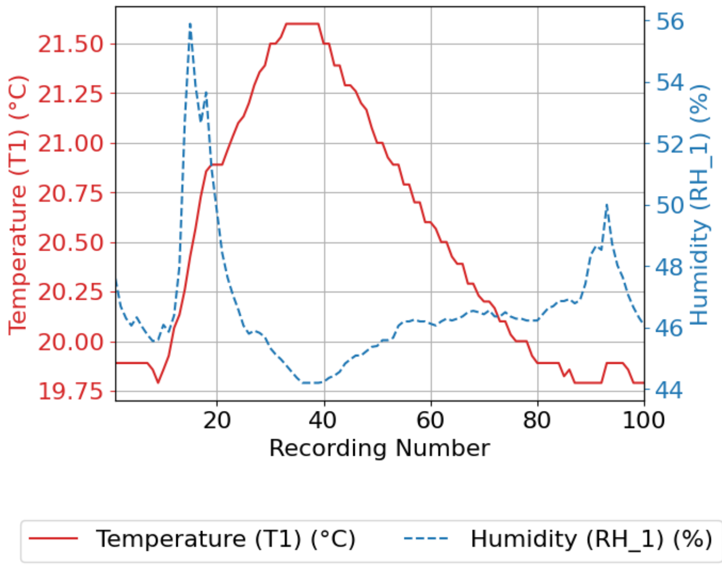

Figure 3. Sample of 100 sequential temperature and humidity readings from real-time sensors.

The data acquisition and preparation process encompassed the following steps (bypassing the framework data collection and communication layers, owing to direct data availability):

1. Retrieved the "Appliances Energy Prediction Data Set" from the UCI Machine Learning Repository.
2. Downloaded the dataset in .csv format.
3. Extracted relevant columns, such as T1 (temperature in the living room) and RH_1 (relative humidity in the living room), along with data from other monitored spaces.
4. Stored the dataset locally in a directory titled Article_dataset for experimental use.

The framework processed these data streams, where each sensor generated readings every second, including the timestamp, temperature (in degrees Celsius °C), and humidity (percentage %). This setup enables real-time ingestion, processing, and analysis across several experimental scenarios to compare different messaging queues, data processing frameworks, and application architectures.

As shown in Figure 4, three experimental campaigns were conducted: the first (left) evaluation of Kafka and RabbitMQ for data ingestion, the second (middle) comparison of Spark Streaming with Apache Hadoop for data processing and analytics, and the third (right) assessment of microservice versus monolithic application architectures.

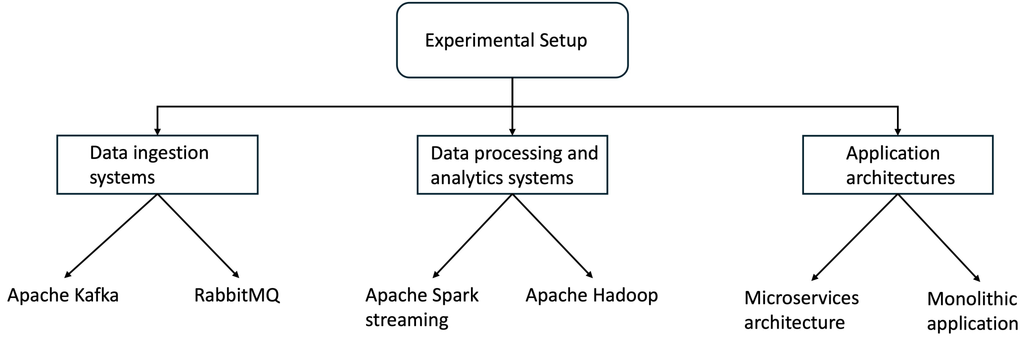

Figure 4. Experimental setup outlining the operational workflow for the considered case study.

4.2.1. Data Ingestion Systems

To assess the data ingestion layer, Apache Kafka and RabbitMQ, which are widely used message queue systems, were deployed in three-node clusters and configured with a replication factor of three for enhanced scalability and fault tolerance. Varying message loads (100, 1000, and 10,000 messages per second) were simulated to reflect different sensor densities in the CM environment. Apache Kafka was set up with temperature and humidity sensors as data producers and data processing systems as consumers. Its distributed fault-tolerant design supports continuous high-volume data streams, which are essential for the CM. The performance metrics for each component were analyzed under various load conditions to confirm the robustness and efficiency of the proposed framework. Conversely, RabbitMQ was configured with sensor data as publishers and consumer applications subscribing to queues using a direct exchange routing configuration to benchmark performance alongside Kafka. Metrics such as the throughput, task duration, and scalability were evaluated across the tested message rates.

4.2.2. Data Processing and Analytic Systems

The tested data processing frameworks included both Apache Spark Streaming and Hadoop, with the former configured with a 5 s sliding window and a 1 s batch interval to compute rolling averages for temperature and humidity. In-memory processing enables low-latency responses for real-time decision making in manufacturing.

Apache Hadoop was used as a batch-processing baseline for calculating temperature and humidity averages. Although it has a higher latency than Spark Streaming, it serves as a reference point for assessing real-time capabilities. The analyzed metrics included the processing latency, resource utilization, and scalability under increased data loads.

4.2.3. Application Architectures

Independent task-based microservices were containerized with Docker and orchestrated with Kubernetes, allowing dynamic scaling and fault isolation, and enabling the scaling of individual services according to load demands. By contrast, a monolithic integrated Flask application manages all tasks (data ingestion, processing, and storage) within one instance. Although simpler to deploy, it lacks fault isolation and scalability, proving that it is more resource-intensive under high loads. The comparison metrics include the response time for request processing, throughput, and scalability.

4.2.4. Experimental Setup

All experiments were conducted on a system with an Intel Core™ i7-6 core processor (Intel Corporation, Santa Clara, CA, USA) and 16 GB RAM. The software versions used were as follows:

- Apache Kafka: 3.8.1.
- RabbitMQ: 4.0.2.
- Apache Spark Streaming: 3.5.2.
- Apache Hadoop: 3.4.1.
- Flask: 3.0.3.
- Docker: 26.0.2.
- Kubernetes: 1.31.2.

The performance metrics for each component were analyzed under various load conditions to confirm the robustness and efficiency of the proposed framework.

5. Results

This section presents an analysis of the performance comparison between Apache Kafka and RabbitMQ in data ingestion systems. This study utilized custom-designed producer and consumer algorithms tailored to each system architecture to evaluate their efficiency under varying message loads. Kafka was integrated with Zookeeper for resource management, while RabbitMQ was deployed using Kubernetes within a Docker environment.

5.1. Data Ingestion Systems

Table 1 summarizes the comparative performance of Apache Kafka and RabbitMQ under various message loads, utilizing both producer (Algorithms A1 and A3) and consumer (Algorithms A2 and A4) components. Kafka was configured with Zookeeper for resource management, while RabbitMQ was orchestrated using Kubernetes within Docker.

The algorithms were optimized specifically for Kafka and RabbitMQ to enhance throughput, latency, and durability by leveraging the architectural advantages of each system. Kafka producer (Algorithm A1) dynamically manages message loads through an adjustable message count parameter (`max_messages`), along with a logging mechanism for progress tracking, which is particularly valuable for handling high data volumes in CM environments. The consumer algorithm (Algorithm A2) focuses on sequential processing with robust error-handling, which is beneficial for applications that require strict data ordering.

For RabbitMQ, the producer (Algorithm A3) employed in-memory messaging to maintain a low-latency dispatch suitable for real-time applications. The consumer

(Algorithm A4) incorporates a callback mechanism for immediate message processing and a prefetch count of one to facilitate rapid adaptability under a dynamic workload. This setup supports high-speed message handling although it does not enforce strict message ordering. The performance metrics for Kafka and RabbitMQ are listed in Table 1.

The Kafka setup demonstrated particular strengths in managing high-volume data streams by leveraging the dynamic message load management and progress-tracking features of the producer algorithm. These abilities make Kafka well suited for CM environments that deal with large-scale data processing. Kafka consumer algorithm emphasizes sequential processing with robust error handling, catering to applications that require strict data ordering. In contrast, the RabbitMQ configuration exhibits advantages in real-time, low-latency applications. The producer algorithm uses in-memory message management for swift dispatch, whereas the consumer algorithm employs a callback mechanism that enables rapid adaptation to changing workloads. This setup allows RabbitMQ to excel in high-speed message-handling scenarios without strict message ordering provided by Kafka.

Table 1. Performance comparison of Apache Kafka and RabbitMQ across varying message loads.

Messages Processed	System Component	Time (s)	Throughput (msg/s)
100	Kafka Producer	0.16	644.87
	Kafka Consumer	2.21	45.28
	RabbitMQ Producer	0.16	644.83
	RabbitMQ Consumer	0.06	1614.62
1000	Kafka Producer	1.40	714.21
	Kafka Consumer	0.47	2137.99
	RabbitMQ Producer	1.49	671.55
	RabbitMQ Consumer	0.46	2161.49
10,000	Kafka Producer	14.02	713.25
	Kafka Consumer	5.27	1895.80
	RabbitMQ Producer	15.97	626.34
	RabbitMQ Consumer	4.61	2169.57

The results of this study, which are graphically represented in Figure 5, demonstrate higher throughput values for Kafka producers than RabbitMQ producers across varying user counts. These findings suggest that Kafka exhibits superior performance in managing structured, sequence-sensitive data, particularly in producer operations, thereby making it more suitable for applications that require ordered message handling. Conversely, RabbitMQ demonstrated higher consumer throughput across all message rates, highlighting its efficacy in real-time high-throughput scenarios where immediate message consumption is prioritized over strict message ordering.

5.2. Data Processing and Analytic Systems

After the evaluation of the two data ingestion systems, a detailed assessment was conducted on the performance of two data processing systems: Spark Streaming and Hadoop Batch Processing. The Kafka producer script was executed with a maximum message count of 10,000, leveraging its favorable throughput and processing time results compared to RabbitMQ under similar input conditions (see Table 1). The comparative performance metrics for Spark Streaming and Hadoop Batch Processing are summarized in Table 2, which serves as a critical reference for analyzing their suitability within the CM framework.

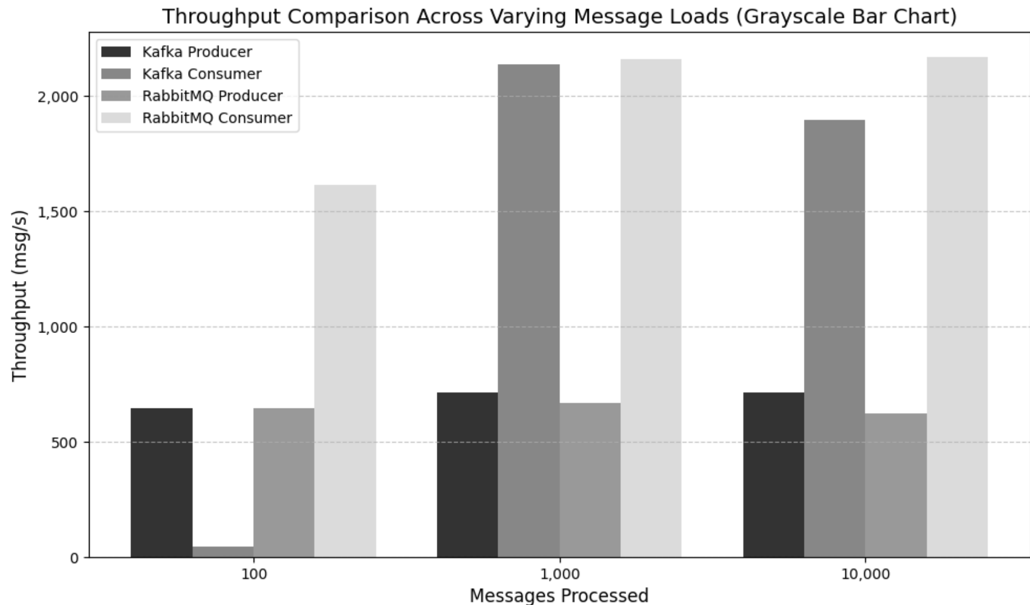

Figure 5. Bar chart of performance comparison of Apache Kafka and RabbitMQ across varying message loads.

Table 2. Performance comparison of Spark Streaming and Hadoop batch processing systems.

Metric	Spark Streaming	Hadoop Batch
Batch ID 0 Record Count	103	-
Batch ID 1 Record Count	2324	-
Batch ID 2 Record Count	161	-
Total Records Processed	2588	10,000
Total Processing Time (s)	4.236	5.976
Throughput (records/s)	612.25 (avg)	1673.45
Average Latency (s)	1.390	-
CPU Usage (Before)	33.5%	28.6%
CPU Usage (After)	39.4% (avg)	33.1%
Memory Usage (Before)	64.1%	63.5%
Memory Usage (After)	64.0% (avg)	63.8%
Average Temperature (avg_T1)	21.68	21.67
Average Relative Humidity (avg_RH_out)	82.43	79.77

An analysis of the metrics presented in Table 2 demonstrates their pivotal role in substantiating the advantages of the proposed framework, as follows:

- Total Records Processed: While Hadoop Batch Processing handles a larger volume of records (10,000 compared to 2588 for Spark Streaming), the discrepancy arises from the batch-oriented nature of Hadoop versus the real-time, continuous nature of Spark Streaming. This distinction reflects the suitability of Spark Streaming for low-latency CM applications, in which immediate insights are prioritized over total volume.
- Total Processing Time (s): Spark Streaming outperforms Hadoop Batch Processing with a shorter processing time (4.236 s vs. 5.976 s). This metric underscores the efficiency of Spark Streaming's in-memory computation model, which reduces the time required to derive actionable insights, and is an essential feature for dynamic and fast-paced manufacturing environments.
- Throughput (records/s): While Hadoop exhibits higher throughput (1673.45 records/s compared to 612.25 records/s of Spark), this metric aligns with its design for batch-

processing scenarios. The relatively low throughput of Spark is a trade-off for its capability to provide real-time processing and immediate feedback.

- Average Latency (s): The 1.390 s average latency of Spark Streaming is a critical metric that highlights its superiority in delivering near-instantaneous data processing. Hadoop, which is inherently batch-oriented, does not support latency measurements in the same sense that its outputs are processed in bulk.
- CPU and Memory Usage: Both systems demonstrate efficient resource utilization, with the average CPU and memory usage of Spark Streaming increasing marginally during operation. This slight increase signifies the ability of Spark to manage resources adaptively in real time, ensuring consistent performance, even under varying workloads.
- Environmental Metrics: The nearly identical values for Average Temperature and Average Relative Humidity across both systems validate the integrity of data processing under similar conditions. However, the real-time aggregation of Spark Streaming offers a temporal advantage for monitoring environmental conditions in CM applications.

The superior performance of Spark Streaming was further bolstered by several innovations embedded in Algorithm A5. Real-time system monitoring enables dynamic adjustments to resource constraints, whereas cumulative tracking of metrics, such as latency and processing times, facilitates iterative process optimization. These features make Spark Streaming an indispensable tool for scenarios that demand continuous, low-latency data processing, operational efficiency, and rapid decision making in CM environments. Conversely, the high throughput of Hadoop Batch Processing remains valuable for prioritizing large-scale data aggregation and offline analytics. The inclusion of enhancements such as batch size optimization and metric logging ensures that Hadoop remains relevant for traditional high-volume data processing needs, albeit at the expense of real-time responsiveness. By integrating these insights, we emphasize the necessity of balancing throughput and latency metrics when selecting data-processing systems for CM applications. The ability of Spark Streaming to deliver low-latency, real-time insights complements the high-throughput capabilities of Hadoop Batch Processing, underscoring its role in optimizing the performance and scalability of our proposed nine-layer CM framework.

5.2.1. Spark Streaming Innovations

In Algorithm A5, several enhancements are introduced to improve the real-time data-processing capabilities of Spark Streaming:

- Real-time system monitoring: the system tracks CPU and memory usage before and after each batch, enabling efficient real-time resource management. This feature is vital for adjusting manufacturing processes based on available resources.
- Adaptive termination mechanism: the algorithm includes an automatic termination feature that stops processing when no new data are received beyond a defined threshold, optimizing resource utilization and enhancing energy efficiency.
- Enhanced aggregation of environmental data: temperature and humidity readings are averaged over time, providing real-time environmental data that are critical for precise manufacturing control.
- Cumulative metrics tracking: the algorithm logs average latencies, processing times, and resource usage, offering a detailed performance log that aids in refining CM processes or scaling for higher demand.
- Configurable query timeout: a flexible timeout parameter is included, allowing adaptability to various manufacturing needs, whether for continuous operation or on-demand scaling.

5.2.2. Hadoop Batch Processing Innovations

In the Hadoop batch processing system, the following enhancements were applied to optimize performance in a batch-oriented context (refer to Algorithm A6):

- Batch limiting for efficient processing: the system processes a set number of records (10,000) to optimize batch sizes for improved performance in a traditional Hadoop setting.
- Artificial delay for simulated batch processing: a controlled delay simulates the time needed to gather and process large data batches, replicating real-world batch intervals in a manufacturing environment.
- System metrics' tracking pre- and post-processing: CPU and memory usage are monitored before and after processing to assess resource utilization throughout the batch processing cycle.
- Throughput and latency calculation: processing time and throughput are recorded to evaluate the batch system's efficiency.
- Logging of aggregate results and performance metrics: key performance metrics and aggregate results are logged, creating a performance record for further analysis and optimization.

These improvements make both systems more efficient and adaptable for CM applications. The analysis (see Table 2) highlights the low-latency and real-time capabilities of Spark Streaming, which offers distinct advantages over traditional Hadoop batch processing. Although Hadoop is effective for large-scale data processing, it introduces higher latency and lower throughput, making Spark Streaming preferable for scenarios that require immediate data insights.

5.3. Application Architecture

This subsection presents a comparative evaluation of monolithic and microservice architectures tested under varying user loads (100, 1000, and 10,000 users) using the Locust tool (an open-source performance/load testing tool). Key metrics, including the user count, average response time (ms), throughput (requests per second), and total requests, were analyzed to assess the scalability and efficiency of each architecture.

5.3.1. Performance Evaluation for the Monolithic Application

The monolithic application developed in Flask was designed to ingest, process, and store the temperature and humidity data. The application includes the following functionalities:

- Data ingestion: the application reads the temperature_humidity_data.csv file using the Pandas library.
- Data processing: real-time calculation of mean temperature and humidity values facilitates immediate data analysis.
- Performance measurement: response times for data processing requests are tracked and integrated into the data processing function, with results returned in JSON format, allowing for easy client-side consumption.

Performance testing was conducted with Locust under user loads from 100 to 10,000 with Table 3, showing the obtained performance metrics.

Algorithms A7 and A8 detail the core functionalities of the monolithic application and load testing scripts, respectively, providing insights into the system's performance under varying loads.

Table 3. Performance metrics for the monolithic application.

User Count	Average Response Time (ms)	Throughput (Requests/s)	Total Requests
100	223.68	44.31	13,239
1000	9372.87	82.66	24,775
10,000	23,086.79	336.61	102,972

5.3.2. Performance Evaluation for the Microservice Architecture

The microservice-based architecture comprises three independent services: ingestion, processing, and storage. Each service was tested with the same user load conditions as the monolithic architecture. Table 4 presents the response time, throughput, and total requests for each of the three services under different user loads.

Table 4. Performance metrics for the microservices architecture.

Service	User Count	Average Response Time (ms)	Throughput (Requests/s)	Total Requests
Ingestion	100	34.90	32.46	19,388
Ingestion	1000	3.65	329.48	197,354
Ingestion	10,000	11,846.57	607.43	366,940
Processing	100	1.94	32.92	19,670
Processing	1000	5.07	329.95	197,656
Processing	10,000	10,608.47	677.91	407,950
Storage	100	1.92	33.10	19,770
Storage	1000	5.27	330.09	197,786
Storage	10,000	10,488.53	687.86	413,726

The microservice architecture was designed to prioritize scalability and flexibility with independent deployment of each service. The following innovations are implemented to optimize performance (see Algorithms A9–A15).

- Modular microservice design: each service (ingestion, processing, and storage) functions independently, promoting scalability and flexibility within the system.
- Integration of Flask with data processing: Flask is used for data ingestion and processing, supporting real-time responses and improving application responsiveness.
- Dynamic data processing: the system dynamically calculates mean temperature and humidity values, enabling immediate data analysis and insights.
- Performance measurement: real-time response tracking is incorporated into data processing functions, providing feedback on performance impacts directly within the application.
- Error handling: robust error-handling mechanisms ensure graceful failure management, enhancing the application's reliability.
- Realistic load simulation: Locust simulates user behavior with varying loads and dynamic wait times, mimicking real-world interactions for accurate performance measurement.
- Flexible session management: the application dynamically manages user sessions, offering a comprehensive understanding of user interactions during testing.
- Real-time performance metrics: metrics are gathered during load tests, delivering immediate insights into potential bottlenecks and system performance under stress.
- Kubernetes deployment: the architecture is deployed on Kubernetes, supporting high availability and scaling of microservices.

The performance metrics, particularly in terms of throughput for 10,000 user counts, as depicted in Figure 6, demonstrate higher values for the three microservices compared

to that for the monolithic application, indicating that the design enhancements exemplify the capacity of the microservice architecture to manage increased loads more efficiently than the monolithic system while providing enhanced flexibility and scalability for future expansions.

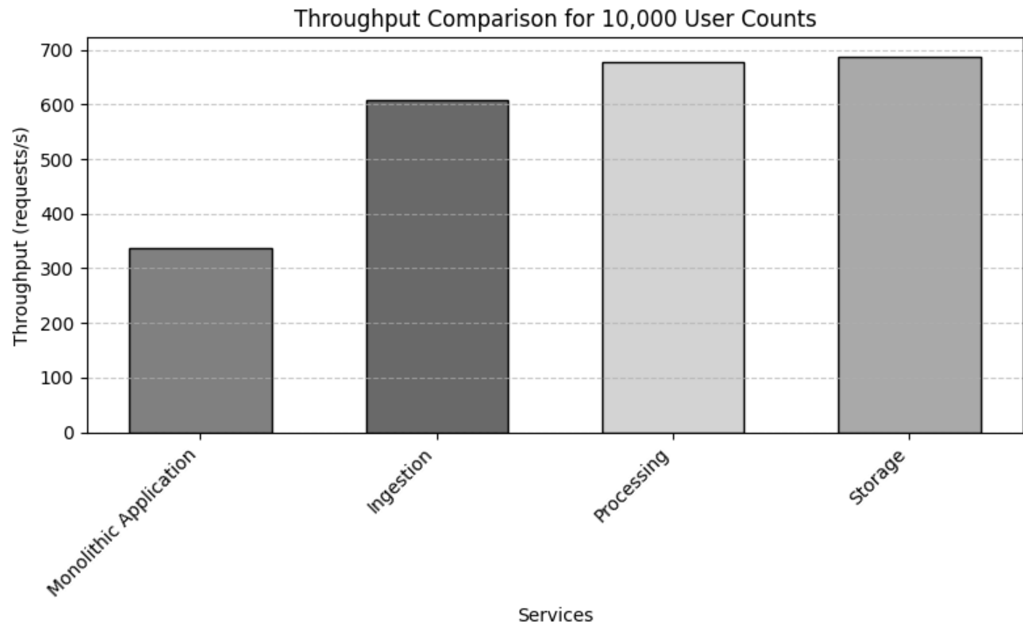

Figure 6. Bar chart of performance metrics for the Monolithic app vs. Microservice architecture.

6. Discussion

The proposed nine-layer framework was designed to address the challenges of scalability, resource orchestration, and data security in modern decentralized manufacturing settings. For this implementation, we use technologies that excel in handling the ingestion and processing of large-scale IoT sensor data with low latency, thereby ensuring real-time responsiveness and long-term operational reliability. This is especially critical for applications in CM and AM, where the ability to monitor and control processing parameters such as temperature and humidity is essential for ensuring the quality and consistency of 3D-printed parts produced through filament material extrusion.

The experimental campaigns clearly demonstrated the performance gains achieved through the integration of Apache Kafka for data ingestion, Spark Streaming for processing, and a microservice architecture for overall framework scalability and resilience. Specifically, the performance evaluation of Kafka and RabbitMQ (Table 1) highlighted the superior capabilities of Kafka in high-throughput, low-latency data ingestion scenarios, which are critical for CM and AM applications. The log-based architecture of Kafka, combined with its durability and fault tolerance, makes it particularly well-suited for handling continuous streams of sensor data generated in AM workflows. For example, Kafka achieved a throughput of 714.21 messages per second for a workload of 1000 messages, outperforming the 671.55 messages per second of RabbitMQ, under the same conditions. This superior throughput, coupled with the consistent performance of Kafka under increased data volumes, ensures that the framework can handle the large-scale datasets required for real-time monitoring of processing parameters such as temperature and humidity in AM. Such capabilities enable immediate fault detection and corrective action, reduce the risk of defects in 3D-printed parts, and enhance the overall operational efficiency.

Similarly, the adoption of Spark Streaming for real-time processing has demonstrated significant advantages over traditional batch processing systems such as Hadoop, as shown

in Table 2. The micro-batch architecture of Spark allows for the near-instantaneous ingestion and analysis of IoT sensor data, achieving substantial reductions in processing time and latency. In our experiments, Spark processed equivalent workloads in 4.236 s compared with 5.976 s obtained with Hadoop, achieving a lower average latency of 1.390 s. This real-time processing capability is particularly advantageous for CM and AM scenarios, which require rapid responsiveness to environmental changes during the production of 3D-printed parts. By leveraging Spark's ability to process large volumes of streaming sensor data in real-time, the framework can quickly identify inefficiencies or faults in AM systems, thereby ensuring optimal parameter control and product quality.

The use of microservice architecture further enhances the ability of the framework to support high-throughput, low-latency operations across its layers, as indicated by the comparative results in Tables 3 and 4. Microservices demonstrate superior scalability and fault tolerance under an increasing workload compared with monolithic systems. For instance, under a 10,000 user load, the microservice-based ingestion service processed 366,940 requests compared to the 102,972 requests of the monolithic system. Additionally, microservices maintained sub-10 ms response times for ingestion, processing, and storage tasks, while the monolithic system exhibited response times exceeding 9000 ms. This modularity ensures that failures are isolated into specific components, thereby maintaining the functionality of other parts of the framework during partial disruptions. The scalability of the microservice architecture allows the framework to dynamically adapt to the demands of IoT sensor data streams, which is a critical requirement for CM and AM applications where uninterrupted data flow and rapid analytics are essential.

By integrating Kafka, Spark Streaming, and a microservice architecture, the proposed nine-layer framework delivers unparalleled performance gains for modern industrial applications. The high-throughput durability of Kafka supports seamless data flow across real-time and archival tiers, whereas the in-memory processing of Spark Streaming provides actionable insights in near real time. The microservice architecture ensures that the framework remains robust and scalable, and is capable of handling both real-time alerts and long-term resource planning tasks. These combined features make the framework highly adaptable to the evolving needs of CM, with particular emphasis on the AM sector. By enabling precise control over critical manufacturing parameters, such as temperature and humidity, during the filament extrusion process, the framework ensures the superior quality and consistency of the 3D-printed parts.

Therefore, we demonstrated the effectiveness of the nine-layer framework in addressing the complexities of modern industrial ecosystems. The performance gains achieved through Kafka, Spark Streaming, and microservices underscore the framework's potential to transform decentralized manufacturing systems. Future work could focus on extending the applicability of the framework by incorporating advanced predictive analytics and optimizing cross-tier communication, further validating its role as a transformative solution for CM and AM.

7. Conclusions

Considering the data ingestion processing and real-time analytics in CM, our study confirms that the log-based architecture of the Kafka platform is optimal for robust, scalable, and durable data ingestion. The high throughput and ordered message delivery of the Kafka platform make it particularly effective for real-time monitoring and predictive maintenance, whereas RabbitMQ remains suitable for low-latency small-scale tasks. For data processing, Spark Streaming outperforms Hadoop in latency-sensitive applications, leveraging its in-memory and micro-batch architecture to deliver real-time analytics and actionable

insights. This responsiveness is essential for live monitoring and adaptive manufacturing operations, making Spark a preferred solution for dynamic CM environments.

Our analysis also highlights the scalability and efficiency of microservices over monolithic architectures. The modular structure of the microservices supports independent deployment, dynamic scaling, and fault tolerance, thereby enabling superior performance under variable loads. Containerization with Docker and orchestration via Kubernetes further enhance flexibility and resilience, ensuring reliable operations in distributed manufacturing scenarios. These choices collectively form a robust and scalable foundation for real-time CM applications, and they are validated through experimental campaigns that demonstrate high throughput, low latency, and adaptability.

Future research should focus on deploying this framework in industrial case studies to optimize predictive maintenance, real-time quality control, and resource allocation. Incorporating advanced security measures, such as encryption, Role-Based Access Control, and intrusion detection, is critical for addressing cybersecurity challenges inherent in shared CM environments. Techniques such as blockchain and AI-driven threat intelligence can enhance data integrity and resilience against cyber threats, particularly in sensitive applications, such as AM. Additionally, integrating machine learning models for predictive analytics and adaptive system configurations could further optimize performance and responsiveness in real-world scenarios.

In conclusion, the proposed framework integrates Kafka, Spark Streaming, and microservices to address critical CM requirements and achieve high performance, scalability, and resilience. This design establishes a strong foundation for advanced industrial applications with opportunities for future enhancements in security, adaptability, and real-world testing. These advancements have ensured that the framework remains at the forefront of reliable and efficient CM systems in evolving manufacturing environments.

Author Contributions: Conceptualization, M.P.; methodology, M.P. and A.P.; validation, M.P. and A.P.; resources, G.P. and E.F.; writing—original draft preparation, M.P. and A.P.; writing—review and editing, M.P. and A.P.; supervision, M.P and G.P.; project administration, G.P. and E.F.; funding acquisition, G.P. and E.F. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by Puglia Region (Italy)—Project "SMILE&SCALETECH" (P.O. PUGLIA FESR 2014-2020).

Data Availability Statement: The data analyzed in this manuscript are available at https://archive.ics.uci.edu/dataset/374/appliances+energy+prediction (accessed on 1 January 2025).

Acknowledgments: The authors express their gratitude for the administrative and technical assistance provided by the company Advantech S.r.l. in Lecce, Italy.

Conflicts of Interest: Author E.F. was employed by the company Laser Romae S.r.l. in Rome, Italy. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest. The funder had no role in the design of this study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; nor in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

AM	Additive Manufacturing
CC	Cloud Computing
CM	Cloud Manufacturing
IoT	Internet of Things
RBAC	Role-Based Access Control

Appendix A

In this appendix section, we presented a series of innovative pseudocodes corresponding to key layers of the proposed framework.

Algorithm A1 outlines the pseudocode for the description of the Kafka producer considering varying input workloads.

Algorithm A1 Kafka Producer Pseudocode

```
Require: Kafka and essential libraries (pandas, json, etc.)
Ensure: Message count, total time, and throughput metrics
  Initialize logger with INFO level and load dataset as data
  for max_messages in {100, 1000, 10000} do
    Sample data_sample of size max_messages from data
    Initialize Kafka producer and start_time
    for each row in data_sample up to max_messages do
      Produce message to 'temp_humidity_topic'; increment message_count
      Log progress every 10 messages; sleep 0.001s for simulation
    end for
    Flush producer, calculate total_time and throughput as message_count / total_time
    Log final metrics: message_count, total_time, and throughput
  end for
```

Algorithm A2 reports the pseudocode defining the Kafka consumer tool, still considering varying numbers of input messages.

Algorithm A2 Kafka Consumer Pseudocode

```
Require: Kafka and essential libraries (json, time, etc.)
Ensure: Processed message count, total time, and throughput metrics
  Initialize logger with INFO level and set consumer_config with Kafka parameters
  Initialize Kafka consumer with consumer_config and subscribe to 'temp_humidity_topic'
  function CONSUME_MESSAGES(max_messages)
    Start timer (start_time); initialize message_count to 0
    while message_count < max_messages do
      Poll for messages with timeout
      if message error then
        Log error and continue polling
      else
        Decode JSON message, log message, and increment message_count
        if message_count is divisible by 10 then
          Log progress
        end if
      end if
    end while
    Calculate total_time and throughput as message_count / total_time
    Log final metrics and close consumer
  end function
  Run consume_messages for max_messages values {100, 1000, 10000}
```

Furthermore, we described the pseudocodes for, respectively, configuring both RabbitMQ producer and consumer with varying input workloads (Algorithms A3 and A4).

Algorithm A3 RabbitMQ Producer Pseudocode

```
Ensure: Total messages sent, total time, and throughput metrics
  Initialize logger with INFO level
  Load dataset as data
  for max_messages in {100, 1000, 10000} do
      Sample data with max_messages rows as data_sample
      Establish RabbitMQ connection and declare durable queue 'temp_humidity_queue'
      Start timer (start_time) and log initial message
      Initialize message_count to 0
      for each row in data_sample do
          Convert row to json and publish to RabbitMQ queue 'temp_humidity_queue'
          Increment message_count and sleep briefly
          if message_count is divisible by 10 then
              Log progress
          end if
      end for
      Close connection, calculate total_time, and compute throughput
      Log final metrics and output:
        "Total Messages: message_count", "Total Time: total_time", "Throughput:
  throughput"
  end for
```

Algorithm A4 RabbitMQ Consumer Pseudocode

```
Ensure: Total messages consumed, total time, and throughput metrics
  Initialize logger with INFO level
  Establish RabbitMQ connection and declare durable queue 'temp_humidity_queue'
  for max_messages in {100, 1000, 10000} do
      Set start_time and initialize message_count to 0
      function CALLBACK(ch, method, properties, body)
          Parse body as message_data; log data and acknowledge message
          Increment message_count
          if message_count is divisible by 10 then
              Log progress
          end if
          if message_count ≥ max_messages then
              Stop consuming
          end if
      end function
      Log start; set prefetch to 1; register callback and start consuming
      On exit, calculate total_time and compute throughput
      Log final metrics and output:
        "Total Messages: message_count", "Total Time: total_time", "Throughput:
  throughput"
      Close connection
  end for
```

The pseudocode for configuring the Data Processing and Analytics layer is detailed in Algorithm A5, with emphasis on the Spark Streaming tool.

Algorithm A5 Spark Streaming Code for Real-Time Monitoring

```
  Imports and Initial Setup
  Import necessary libraries:
    - Spark SQL libraries
    - System monitoring and time libraries
  Define data schema for incoming temperature and humidity data
```

Algorithm A5 *Cont.*

Initialization
Start Spark Session for real-time streaming and configure it to read from Kafka
Stream Kafka data by:
- Connecting to Kafka server
- Subscribing to the specified topic
- Loading data stream

Parse and structure data by:
- Converting data to string
- Parsing JSON data based on schema

Define Termination and Metrics Variables
Set termination parameters for detecting a halt in new data
Initialize cumulative metrics dictionary to track performance and data averages over time
Processing Function: Process Each Micro-Batch
function PROCESS_BATCH(`batch_df`, `batch_id`)
    Record start time of processing
    Capture CPU and memory usage before batch processing
    Count records in the batch
    if no new data detected after multiple batches **then**
        Increment empty batch counter
        if counter exceeds limit **then**
            Terminate the streaming job
        end if
    else
        Reset empty batch counter
    end if
    Calculate data averages by:
    - Computing average temperature and humidity values for each sensor
    - Storing these averages in a temporary structure for this batch

    Calculate performance metrics by:
    - Computing processing time, throughput, and latency
    - Recording CPU and memory usage after processing

    Update cumulative metrics with batch statistics (latency, temperature, and humidity averages)
    Log performance data by writing batch ID, record count, processing time, and system metrics to a log file
end function
Streaming Job Execution
Configure streaming query to process each batch with `process_batch` by:
- Using `foreachBatch` to apply the function to each micro-batch
- Setting `outputMode` to update with each new batch

Start and await termination of the streaming query with a timeout

Algorithm A6 reports the pseudocode for the Hadoop batch processing system.

Algorithm A6 Batch Processing Framework Using Hadoop

Ensure: Batch processing results and system metrics
Imports and Initial Setup
Import necessary libraries, including Spark SQL and system monitoring libraries
Define data schema for temperature and humidity
Initialize Spark Session
Start a Spark session with Kafka configuration
Configure Kafka settings by:
- Setting `bootstrap_servers`
- Subscribing to the specified topic
- Setting offset to `earliest` to process all data

Algorithm A6 *Cont.*

Data Loading and Batch Limiting
Load data from Kafka in batch mode
Limit data to first 10,000 records for batch simulation
Introduce artificial delay for simulated batch processing
System Metrics Before Processing
Record CPU and memory usage before processing
Start timer to measure batch processing time
Data Processing and Aggregation
Convert batch data to string format
Parse JSON and extract relevant fields according to schema
Calculate average values for each temperature and humidity sensor
Calculate Performance Metrics
Record total processing time
Count records processed and calculate throughput
System Metrics After Processing
Record CPU and memory usage after processing
Log Output to File
Format log output with batch processing results by:
- Total records processed, processing time, throughput
- CPU and memory usage before and after processing
- Calculated averages for sensor data
Write log output to file
Terminate Spark Session
Stop the Spark session

In Algorithm A7, the data processing for temperature and humidity is implemented for the monolithic app and the performance of the monolithic system is further evaluated through a simulation in Algorithm A8.

For the microservices architecture, the ingestion service is described in Algorithm A9, followed by the processing service in Algorithm A10 and storage service in Algorithm A11. Deployment of these services is handled by Kubernetes, as described in Algorithms A12–A14.

Performance testing using Locust is outlined in Algorithm A15.

Algorithm A7 Data Processing for Temperature and Humidity

Require: CSV file path
Ensure: JSON response containing mean temperature, mean humidity, and response time
Initialize `Flask` application ▷ Innovative integration of `Flask` for data processing and web services
function PROCESS_DATA
`start_time` ← `current_time()`
`data` ← `load_CSV(CSV_file_path)` ▷ Efficient CSV file ingestion for real-time processing
`mean_temp` ← `calculate_mean(data, 'temperature')` ▷ Dynamic calculation of mean temperature
`mean_humidity` ← `calculate_mean(data, 'humidity')` ▷ Dynamic calculation of mean humidity
`response_time` ← `current_time()` - `start_time` ▷ Real-time performance metric
`result` ← { `'mean_temperature': mean_temp, 'mean_humidity': mean_humidity, 'response_time': response_time` }
return `JSON(result)` ▷ Response in JSON format for client applications
end function
Error Handling:
if an error occurs **then**
return `JSON({ 'error': error_message })` ▷ Robust error handling for reliable operation

Algorithm A8 Locust User Simulation for Performance Testing

```
Require: Target URL, number of users, spawn rate
Ensure: Performance metrics collected during load test
  function UserBehavior
    Initialize user_session    ▷ Dynamic user session management for flexible testing
    while session is active do
      wait_time ← random_between(1, 3)    ▷ Dynamic wait times to simulate real-world user behavior
      sleep(wait_time)
      response ← send_GET(request_to_url)    ▷ Request simulation to evaluate response times and success rates
    end while
  end function
```

Algorithm A9 Ingestion Service—Data Loading

```
Require: CSV file path
Ensure: Row count of the dataset
  Initialize Flask application    ▷ Modular service for data ingestion via Flask
  function INGEST_DATA
    Load CSV file    ▷ Efficient and flexible data loading mechanism
    return success_message and row_count
  end function
  Error Handling:
   if an error occurs then
     return error_message    ▷ Robust error handling for seamless operation
  Run Flask app on host 0.0.0.0, port 5001  ▷ Microservice accessibility at defined port
```

Algorithm A10 Processing Service—Data Computation

```
Require: JSON data containing temperature and humidity fields
Ensure: Average temperature and humidity
  Initialize Flask application    ▷ Decoupled microservice for data computation
  function PROCESS_DATA
    Convert JSON data to DataFrame    ▷ Structured data processing for computation
    Compute average temperature from temperature fields    ▷ Real-time analytics on temperature data
    Compute average humidity from humidity fields ▷ Real-time analytics on humidity data
    return results as JSON    ▷ JSON response format for easy integration with client applications
  end function
  Error Handling:
   if an error occurs then
     return error message    ▷ Error handling for enhanced reliability
  Run Flask app on host 0.0.0.0, port 5002
```

Algorithm A11 Storage Service—Data Storage and Retrieval

```
Require: Processed data as JSON
Ensure: Stored data in JSON file, retrieve stored data
  Initialize Flask application    ▷ Microservice to handle data persistence
  function STORE_DATA
    Write data to file    ▷ Innovative approach for storing data in flexible JSON format
    return success message
  end function
```

Algorithm A11 *Cont.*

Error Handling for Storage:
  if an error occurs **then**
    return `error message` ▷ Error handling for smooth data storage
function RETRIEVE_DATA
  Read data from file ▷ Efficient retrieval mechanism for accessing stored data
  return data as JSON
end function
Error Handling for Retrieval:
  if an error occurs **then**
    return `error message` ▷ Graceful error handling for data retrieval
Run `Flask` app on host `0.0.0.0`, port `5003`

Algorithm A12 Ingestion Service—Kubernetes Deployment

Require: Docker image `ingestion_service:latest`
Ensure: Deployment and service for ingestion microservice
  Define Deployment with replicas = 3 ▷ Scalable microservice deployment for high availability
  Use Docker image `ingestion_service`
  Expose port `5001` inside the container
  Define `LoadBalancer` service for external access ▷ Easy integration into Kubernetes architecture
  Map port `5001` to the service

Algorithm A13 Processing Service—Kubernetes Deployment

Require: Docker image `processing_service:latest`
Ensure: Deployment and service for processing microservice
  Define Deployment with replicas = 3
  Use Docker image `processing_service`
  Expose port `5002` inside the container
  Define `LoadBalancer` service for external access
  Map port `5002` to the service

Algorithm A14 Storage Service—Kubernetes Deployment

Require: Docker image `storage_service:latest`
Ensure: Deployment and service for storage microservice
  Define Deployment with replicas = 3
  Use Docker image `storage_service`
  Expose port `5003` inside the container
  Define `LoadBalancer` service for external access
  Map port `5003` to the service

Algorithm A15 Locust Performance Testing

Require: Microservice endpoints
Ensure: Response time and success rate measurements
  Define `IngestionServiceUser` ▷ Simulates user interaction with ingestion service
  Task Test ingestion service with GET request
  Define `ProcessingServiceUser` ▷ Simulates user interaction with processing service
  Task Test processing service with GET request
  Define `StorageServiceUser` ▷ Simulates user interaction with storage service
  Task Test storage service with GET request

References

1. Gharibvand, V.; Kolamroudi, M.K.; Zeeshan, Q.; Çınar, Z.M.; Sahmani, S.; Asmael, M.; Safaei, B. Cloud based manufacturing: A review of recent developments in architectures, technologies, infrastructures, platforms and associated challenges. *Int. J. Adv. Manuf. Technol.* **2024**, *131*, 93–123. [CrossRef]
2. Hayyolalam, V.; Pourghebleh, B.; Chehrehzad, M.R.; Pourhaji Kazem, A.A. Single-objective service composition methods in cloud manufacturing systems: Recent techniques, classification, and future trends. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e6698. [CrossRef]
3. Brecko, A.; Kajati, E.; Koziorek, J.; Zolotova, I. Federated learning for edge computing: A survey. *Appl. Sci.* **2022**, *12*, 9124. [CrossRef]
4. Rashidifar, R.; Bouzary, H.; Chen, F.F. Resource scheduling in cloud-based manufacturing system: A comprehensive survey. *Int. J. Adv. Manuf. Technol.* **2022**, *122*, 4201–4219. [CrossRef]
5. Liu, S.; Cheng, H. Manufacturing Process Optimization in the Process Industry. *Int. J. Inf. Technol. Web Eng. (IJITWE)* **2024**, *19*, 1–20. [CrossRef]
6. Ye, Y.; Wang, M.; Yao, S.; Jiang, J.N.; Liu, Q. Big data processing framework for manufacturing. *Procedia CIRP* **2019**, *83*, 661–664. [CrossRef]
7. Wang, J.; Antwi-Afari, M.F.; Tezel, A.; Antwi-Afari, P.; Kasim, T. Artificial Intelligence in Cloud Computing technology in the Construction industry: A bibliometric and systematic review. *J. Inf. Technol. Constr.* **2024**, *29*, 480–502. [CrossRef]
8. Ge, K.; Nguyen, P.; Arnaout, R. *lucie*: An Improved Python Package for Loading Datasets from the UCI Machine Learning Repository. *arXiv* **2024**, arXiv:2410.09119.
9. Kekevi, U.; Aydın, A.A. Real-time big data processing and analytics: Concepts, technologies, and domains. *Comput. Sci.* **2022**, *7*, 111–123. [CrossRef]
10. Chiappa, S.; Videla, E.; Viana-Céspedes, V.; Piñeyro, P.; Rossit, D.A. Cloud manufacturing architectures: State-of-art, research challenges and platforms description. *J. Ind. Inf. Integr.* **2023**, *34*, 100472. [CrossRef]
11. Raptis, T.P.; Passarella, A. A survey on networked data streaming with apache kafka. *IEEE Access* **2023**, *11*, 85333–85350. [CrossRef]
12. Ooi, B.Y.; Lee, W.K.; Shubert, M.J.; Ooi, Y.W.; Chin, C.Y.; Woo, W.H. A flexible and reliable internet-of-things solution for real-time production tracking with high performance and secure communication. *IEEE Trans. Ind. Appl.* **2023**, *59*, 3121–3132. [CrossRef]
13. Kaur, J. Streaming Data Analytics: Challenges and Opportunities. *Int. J. Appl. Eng. Technol.* **2023**, *5*, 10–16.
14. Chen, W.; Milosevic, Z.; Rabhi, F.A.; Berry, A. Real-time analytics: Concepts, architectures and ML/AI considerations. *IEEE Access* **2023**, *11*, 71634–71657 . [CrossRef]
15. Xu, C.; Du, X.; Fan, X.; Giuliani, G.; Hu, Z.; Wang, W.; Liu, J.; Wang, T.; Yan, Z.; Zhu, J.; et al. Cloud-based storage and computing for remote sensing big data: A technical review. *Int. J. Digit. Earth* **2022**, *15*, 1417–1445. [CrossRef]
16. Böhm, S.; Wirtz, G. Cloud-edge orchestration for smart cities: A review of kubernetes-based orchestration architectures. *EAI Endorsed Trans. Smart Cities* **2022**, *6*, e2. [CrossRef]
17. Al Qassem, L.M.; Stouraitis, T.; Damiani, E.; Elfadel, I.M. Containerized microservices: A survey of resource management frameworks. *IEEE Trans. Netw. Serv. Manag.* **2024**, *21*, 3775–3796. [CrossRef]
18. Pontarolli, R.P.; Bigheti, J.A.; de Sá, L.B.R.; Godoy, E.P. Microservice-oriented architecture for industry 4.0. *Eng* **2023**, *4*, 1179–1197. [CrossRef]
19. Yang, H.; Ong, S.; Nee, A.Y.; Jiang, G.; Mei, X. Microservices-based cloud-edge collaborative condition monitoring platform for smart manufacturing systems. *Int. J. Prod. Res.* **2022**, *60*, 7492–7501. [CrossRef]
20. Lathkar, M. *Building Web Apps with Python and Flask: Learn to Develop and Deploy Responsive RESTful Web Applications Using Flask Framework (English Edition)*; BPB Publications: New Delhi, India, 2021.
21. Tantrapirom, K.; Pungrasmi, P.; Tangthavonsirikul, S.; Jitnaknan, T. A Reliable Real-Time Web Interface for an Online Laboratory. In *Online Engineering and Society 4.0: Proceedings of the 18th International Conference on Remote Engineering and Virtual Instrumentation*; Springer: Berlin/Heidelberg, Germany, 2021; Volume 298, p. 35.
22. Li, D.; Han, D.; Crespi, N.; Minerva, R.; Sun, Z. Fabric-scf: A blockchain-based secure storage and access control scheme for supply chain finance. *arXiv* **2021**, arXiv:2111.13538.
23. Cains, M.G.; Flora, L.; Taber, D.; King, Z.; Henshel, D.S. Defining cyber security and cyber security risk within a multidisciplinary context using expert elicitation. *Risk Anal.* **2022**, *42*, 1643–1669. [CrossRef] [PubMed]
24. Zafar, M.Q.; Khan, M.B.; Zhao, H. Cyber Security in Additive Manufacturing. In *Integration of Heterogeneous Manufacturing Machinery in Cells and Systems*; CRC Press: Boca Raton, FL, USA, 2024; pp. 151–160.
25. Sriram, G.; Rambabu, S. IMPORTANCE OF CYBERSECURITY IN ADDITIVE MANUFACTURING IN THE ERA OF INDUSTRY 4.0. *Int. J. Manuf. Mater. Mech. (IJMMME)* **2024**, *2940*, 6442.
26. Chen, J.; He, J.; Chen, F.; Lv, Z.; Tang, J.; Li, W.; Liu, Z.; Yang, H.H.; Han, G. Towards General Industrial Intelligence: A Survey on IIoT-Enhanced Continual Large Models. *arXiv* **2024**, arXiv:2409.01207.

27. Wang, J.; Xu, C.; Zhang, J.; Zhong, R. Big data analytics for intelligent manufacturing systems: A review. *J. Manuf. Syst.* **2022**, *62*, 738–752. [CrossRef]
28. Alshareef, H.N. Current development, challenges, and future trends in cloud computing: A survey. *Int. J. Adv. Comput. Sci. Appl.* **2023**, *14*, 329–338. [CrossRef]
29. Liu, C.; Su, Z.; Xu, X.; Lu, Y. Service-oriented industrial internet of things gateway for cloud manufacturing. *Robot.-Comput.-Integr. Manuf.* **2022**, *73*, 102217. [CrossRef]
30. Yang, Z.; Zhen, L. How to optimize service-oriented cloud manufacturing. *J. Oper. Res. Soc.* **2024**, 1–15. [CrossRef]
31. Vemulapalli, G. Architecting for Real-Time Decision-Making: Building Scalable Event-Driven Systems. *Int. J. Mach. Learn. Artif. Intell.* **2023**, *4*, 1–20.
32. Tian, H.; Zhao, C.; Xie, J.; Li, K. Dynamic Operation Optimization of Complex Industries Based on a Data-Driven Strategy. *Processes* **2024**, *12*, 189. [CrossRef]
33. Hossain, M.D.; Sultana, T.; Akhter, S.; Hossain, M.I.; Thu, N.T.; Huynh, L.N.; Lee, G.W.; Huh, E.N. The role of microservice approach in edge computing: Opportunities, challenges, and research directions. *ICT Express* **2023**, *9*, 1162–1182. [CrossRef]
34. Wang, L.; Jiang, Y.X.; Wang, Z.; Huo, Q.E.; Dai, J.; Xie, S.L.; Li, R.; Feng, M.T.; Xu, Y.S.; Jiang, Z.P. The operation and maintenance governance of microservices architecture systems: A systematic literature review. *J. Softw. Evol. Process* **2023**, *35*, e2433. [CrossRef]
35. Lim, M.K.; Xiong, W.; Wang, C. Cloud manufacturing architecture: A critical analysis of its development, characteristics and future agenda to support its adoption. *Ind. Manag. Data Syst.* **2021**, *121*, 2143–2180. [CrossRef]
36. Xu, C.; Du, X.; Fan, X.; Yan, Z.; Kang, X.; Zhu, J.; Hu, Z. A modular remote sensing big data framework. *IEEE Trans. Geosci. Remote Sens.* **2021**, *60*, 1–11. [CrossRef]
37. Vitorino, J.P.; Simão, J.; Datia, N.; Pato, M. IRONEDGE: Stream processing architecture for edge applications. *Algorithms* **2023**, *16*, 123. [CrossRef]
38. Naghib, A.; Jafari Navimipour, N.; Hosseinzadeh, M.; Sharifi, A. A comprehensive and systematic literature review on the big data management techniques in the internet of things. *Wirel. Netw.* **2023**, *29*, 1085–1144. [CrossRef]
39. Ramezani, S.B.; Cummins, L.; Killen, B.; Carley, R.; Amirlatifi, A.; Rahimi, S.; Seale, M.; Bian, L. Scalability, explainability and performance of data-driven algorithms in predicting the remaining useful life: A comprehensive review. *IEEE Access* **2023**, *11*, 41741–41769. [CrossRef]
40. Al-Jumaili, A.H.A.; Muniyandi, R.C.; Hasan, M.K.; Paw, J.K.S.; Singh, M.J. Big data analytics using cloud computing based frameworks for power management systems: Status, constraints, and future recommendations. *Sensors* **2023**, *23*, 2952. [CrossRef]
41. Zhang, T.; Zhao, Y.; Jia, W.; Chen, M.Y. Collaborative algorithms that combine AI with IoT towards monitoring and control system. *Future Gener. Comput. Syst.* **2021**, *125*, 677–686. [CrossRef]
42. Dapkute, A.; Siozinys, V.; Jonaitis, M.; Kaminickas, M.; Siozinys, M. Digital Twin Data Management: Framework and Performance Metrics of Cloud-Based ETL System. *Machines* **2024**, *12*, 130. [CrossRef]
43. Assadian, C.F.; Assadian, F. Data-Driven Modeling of Appliance Energy Usage. *Energies* **2023**, *16*, 7536. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.