Stateful Stream Processing for Digital Twins: Microservice-Based Kafka Stream DSL

Ameer B. A. Alaasam
School of Electrical Engineering and Computer Science
South Ural State University
Chelyabinsk, Russia
https://orcid.org/0000-0002-2084-8899

Gleb Radchenko
School of Electrical Engineering and Computer Science
South Ural State University
Chelyabinsk, Russia
https://orcid.org/0000-0002-7145-5630

Andrey Tchernykh
Computer Science Department CICESE Research Center
Ensenada, Mexico
South Ural State University
Chelyabinsk, Russia
https://orcid.org/0000-0001-5029-5212

Abstract—**Digital Twin is a virtual representation of a technological process or a piece of equipment, that supports monitoring, control and state prediction based on the data, gathered from the sensor networks. To parallelize event processing and produce near-real-time insights over data streams, Digital Twin should be implemented based on an Event-Driven architecture. The Event-Driven architecture is loosely-coupled by its nature. One of the recent possible solutions for loose coupling system is a Microservice approach, a cohesive and independent process that interacts using messages. Stateless behavior is the nature of the microservice, but on the other hand, the vast majority of stream processing in Digital Twin imply stateful operations. Thus, in this paper, we propose a case-study of the possibility to use Apache Kafka Stream API (Kafka stream DSL) to build stateful microservice for real-time manufacturing data analysis. Also, in the presented work we discuss the fulfillment of such requirements as fault tolerance, processing latency, and scalability to support the stateful stream processing in Digital Twins implementation.**

Keywords—Digital Twin, Sensors, Stream processing, Event-Driven, Apache Kafka, Microservice

I. Introduction

The current level of connectivity promise greater speed and increased efficiency of data exchange between technological equipment and computing facilities. Newly embedded sensors with connectivity features move data from the production site to the Cloud [1]. Therefore, academic and engineering communities actively engaging in developing the next generation of sensors-based systems [2]. New data analysis approaches allow improving the performance of a single piece of equipment or a technological process [3]. Such concepts as Industry 4.0 [4], Internet of Things [5], cloud robotics and automation [6] imply the need to use different methods of analyzing data streams from sensor networks [7]. These efforts in sensors research, available communication technologies and extensive development of software in the industrial sphere led to Digital Twin (DT) concept. Digital Twin is a virtual representation of a technological process or a piece of equipment, that supports monitoring, control and state prediction based on the data, gathered from the sensor networks. The concept of DT consists of three main parts [8]: real product in a real space, virtual product in virtual space, and connections that tie them. DT supports the analysis of input data from diverse sensors installed on the real product for actualization and tuning of its virtual state [9]. DT can influence the real product state to enhance the performance or mitigate the risks of unexpected behavior [10], [11].

Systems that use data streams, generated by the sensor networks, to support process control actions, assumed to work well based on the Event-Driven Architecture (EDA). EDA is a software architecture for applications that detect and respond to events, that represent significant changes in the state of system or its environment [12]. By its nature, EDA is extremely loosely coupled and highly distributed [13]. Therefore, any tightly coupled architecture that is based on sequential tasks execution does not meet the requirements of Event-Driven systems [7].

To support the concept of DT, taking into account the need to ensure their work within the EDA, a solution based on the organization of loosely coupled interaction of microservices through data streaming middleware is proposed. The microservice architecture is a cloud application design pattern that implies that the application is divided into a number of small independent services, each of which is responsible for implementing a certain feature.

Microservices can be considered as meta-processes in a Meta-operating system: they are independent, they can communicate with each other using messages and they can be duplicated, suspended or moved to any computational resource and so on [14]. According to [15]–[17], stateless behavior is the nature of the microservice approach and preferred over stateful. Stateless microservice behavior provides such cloud computing benefits as high reliability, high availability, on-demand scalability, load balancing, etc [18]. But when the data streaming sequences processing demands the information about the data form the previous time segments, to solve such tasks as counting, aggregating or windowing, we have limited capabilities for implementation of stateless microservice behavior. Stateful behavior requires complex processes for high reliability and availability support, provides less scalability and demands for complex state managing operations [19]. Therefore, any stateful operation is struggling with the complexity of state management. So, providing an efficient mechanism which ensures the correctness of processing, storing and retrieving the state is an essential process in the stateful operations.

To organize loose-coupled data and event exchange between microservices we can use streaming middleware [2], [20]. This paper presents a study about using Apache Kafka [21] middleware as a central nervous system for data exchange between data sources and microservices to support stateful stream processing in DT development. The study also tests the effectiveness of Kafka streams client library (Kafka Streams DSL) for microservice architecture.

The rest of the paper is organized as follows. In chapter II we consider related works and projects, that aimed at the streaming data processing. In chapter III we provide an overview of the architecture issues for stream processing to support DT development. In Chapter IV we define the case study and its architecture. In Chapter V we show the implementation and deployment detail and results. Chapter VI provides discussion about the results and the capabilities that achieved in the experiment and the capabilities provided in Kafka Stream DSL.

II. Related Works

There is a growing interest in the IoT systems data gathering and processing issues in the research community. The authors of [22] proposed IoT data workloads architecture called "Cyclic architecture" based on a combination of distributed event stream processing systems and data from sensor networks. The cyclic architecture contains three layers:

1) Messaging layer for input/output data and to communicate the underlying layers.

2) Processing layer for event processing.

3) Volatile layer for cashing the updated results.

The Cyclic architecture was proposed to compute short term load-predictions and detect outliers over a stream of high-velocity data from smart sensors which measure power consumption. For the same purpose, the authors of [23] proposed to use Redis [24] as Key/Value database together with STORM [25] which packages the logic of stream-processing into units called "bolts". Authors used a ring buffer-based inter-thread messaging library called LMAX to support data transfer between the bolts.

The authors of [26] implemented the CUmulative SUM (CUSUM) statistical control algorithm in STORM. It supports the detection of anomalies in data series from environmental monitoring. They integrated it with Apache ActiveMQ [27] as a real-time messaging service.

In order to provide real-time analytics over a geospatial data stream of a taxi trip in New York City, the authors of [28] proposed a modular processing engine that consists of three components:

1) *Input processor* that pre-process input data and detects outliers to discard them.

2) *Route processor* that identifies frequent routes.

3) *Profit processor* that supports monitoring of the earnings of taxi drivers related to the starting locations.

Recently, there has been an explosive growth of various solutions and platforms focused on the analysis of data stream processing. For example, Apache Spark [29] general processing engine is designed to perform both batch and stream processing, interactive queries, and machine learning. The largest Spark cluster has 8000 nodes. It has been used to sort 100 TB of data 3X faster than Hadoop MapReduce on 1/10th of the machines.

Another widely used solution is Apache Kafka [21] – a fault-tolerant distributed streaming middleware platform that supports horizontal scalability. Data in Kafka cluster stored in topics. The topic is a category name where messages are published and consumed. Each topic split into one or more partitions. The Kafka cluster consists of one or more brokers that support parallel transfer, storage and replication of data in partitions for high performance, flexibility, scalability and fault tolerance.

III. Stream Processing for Digital Twins

An industrial big data processing system such as DT must comply with the cloud-based data processing paradigm in order to benefit from a pay-as-you-go approach. Such a system must be able to scale up its work on-demand, based on the amount of work that exceeds the available number of virtual machines. At the same time, deploying on a large number of cloud-hosted machines often leads to failures. Thus, DT data processing must be fault-tolerant, enabling rapid recovery. In addition, it is necessary to ensure low latency in response to sensor data collected from physical devices. But it's extremely difficult to provide a near real-time response to events when deploying a system in a public cloud infrastructure.

To monitor the assets of high-tech production equipment and demonstrate the capabilities of the event processing system, we have proposed an approach based on the re-design of the Scientific Workflow (SWF) into smaller, loosely coupled sets of SWFs called Micro-Workflows (MWF) [7], [9]. MWF approach allows independent deployment and increases the potential of horizontal scaling. For example, MWFs provided low-latency response can be deployed near the IoT equipment, while MWF that require intensive processing can be deployed in the public cloud.

Maintaining state in stateful operations requires the ability to identify the source of the input signal to determine what other data has been obtained from that source [30]. Keeping the state inside the computing service is considered a bottleneck, so it is necessary to keep the state in a separate resource, for example, in an external database, external file system, caching service, middleware for message exchange, etc. [33]. But a separate resource for handling the state implies availability of additional server resources, which may include additional computing power, memory and data storage, that can lead to a serious challenge in scalability. Therefore, any stateful operation struggling with the complexity of state management. Thus, the provision of an effective mechanism to ensure the correctness of processing, storage and restoration of the state is an essential process in the stateful operations.

In order to solve the problems described above, we propose to base the DT architecture on the microservice concept [31], [17]. Compared to other architectures, the microservice architecture provides a simplified solution for system lifecycle management tasks such as continuous integration, testing, on-demand scalability, etc [32]. But it is necessary to note that while organizing the operation of microservice systems there are often questions related to such issues as communication optimization and data sharing [33]. To address those issues, as a basis for their interaction, we propose the use of data streaming middleware [2].

Apache Kafka provides a set of native client libraries (for example, Kafka Streams DSL), which can be used to build streaming microservices for data processing for different cases of use. It includes the ability to synchronize the state of the computing process with Kafka topics. This is one of the most effective ways to manage the state, which allows you to restore the state from the Kafka topic even in the case when its local copy is lost due to a failure. Also, an application that uses Kafka Streams DSL can simultaneously perform the functions of a data producer, consumer and processor. Figure 1 shows the general concept of using Kafka as the backbone of the IoT system. It contains the following components:

1) Kafka Cluster: a set of dedicated Kafka brokers that serve as a data exchange backbone.

2) Sensors: provide real-time data on the state of the manufacturing process.

3) Kafka Applications: it is a set of applications built using Kafka API. Kafka supports five APIs that clients can use for application development:

– *Producer API:* send data streams to Kafka topics.

– *Consumer API:* read data streams from Kafka topics.

– *Streams API:* transforming data streams from source topic to destination topic.

– *Connect API:* continually pulls data from a source system into Kafka topic or vice versa.

– *AdminClient API:* managing brokers, topics, and other Kafka cluster objects.

These applications can be used in two basic scenarios:

– They can act as *data adapters* to establish communication with the Kafka cluster for non-Kafka applications or to support the data gathering from sensors.

– They can be implemented as a data-processing *microservice* which consume Kafka messages, process them, and send result back to the Kafka cluster.

4) Non-Kafka Applications: a set of applications that do not support native Kafka API, but used for the implementation of domain logic and data processing, for example, Database systems, Hadoop, etc.

The capabilities offered by Kafka can help to build microservices for data stream processing to support DT development. However, these capabilities can also cause problems. For example, state synchronization with Kafka topics provides reliability for fault tolerance, but on the other hand, can lead to increased latency.

In this paper, we would examine most crustal aspects of Kafka stream API to meet the requirements of developing stateful stream processing microservices to support DT development. We will analyze a case study of solving a real manufacturing challenge in the field of IoT using the Kafka Stream API.

IV. Case-Study Architecture

We have developed a case study to analyze the applicability of Kafka Stream DSL to the development of microservices for real-time data streams processing. This case study provides a parametric study of latency and response time and discusses other parameters such as fault tolerance, scalability and ease of deployment.

A. Data and Processing Operations

This research study uses the real sensors data from DEBS 2012 Grand Challenge: Manufacturing equipment [34]. The delay between two consecutive data points is about 10 ms. The study considers the operators of the first query in the challenge. Source data point includes 66 fields. The first operator's group detects the state change of input fields and emits them along with timestamps of the state change. The second operator's group correlates the change of state of the sensor and the change of state of the valve by calculating the time difference between the occurrence of the state changes and emits them along with timestamps. The goal of the last operators set is to constantly monitor the trend for the calculated values from the previous output data.

B. Case-Study Architecture

To simulate the behavior of real sensors, we have developed a Producer component that reads information from the source data file and sends the data to the source topic in the Kafka Cluster with a delay of about 10 milliseconds between each data point.

Independently of this, the data processing microservice connects to the Kafka source topic, consumes the incoming data, processes it and sends the processing results to the Kafka result topic. Additionally, Kafka provides the possibility of managing the data processing state by synchronizing the local state with the set of intermediate Kafka topics. This microservice is the implementation of the first Debs 2012 query processing process, which contains a set of operators described in the section above.

Figure 2 shows the steps and components of the implementation of data processing within this case. The following key steps of data processing implemented within the framework of the created microservice can be identified:

1. *Consumption* of the source messages.
2. *The first stage of processing* is responsible for the detection of a change in the state of the input signals and their further transmission together with timestamps of the state change. The following significant processing steps were implemented for this purpose:
   a. *Grouping the data stream*: each group is responsible for different data sources.

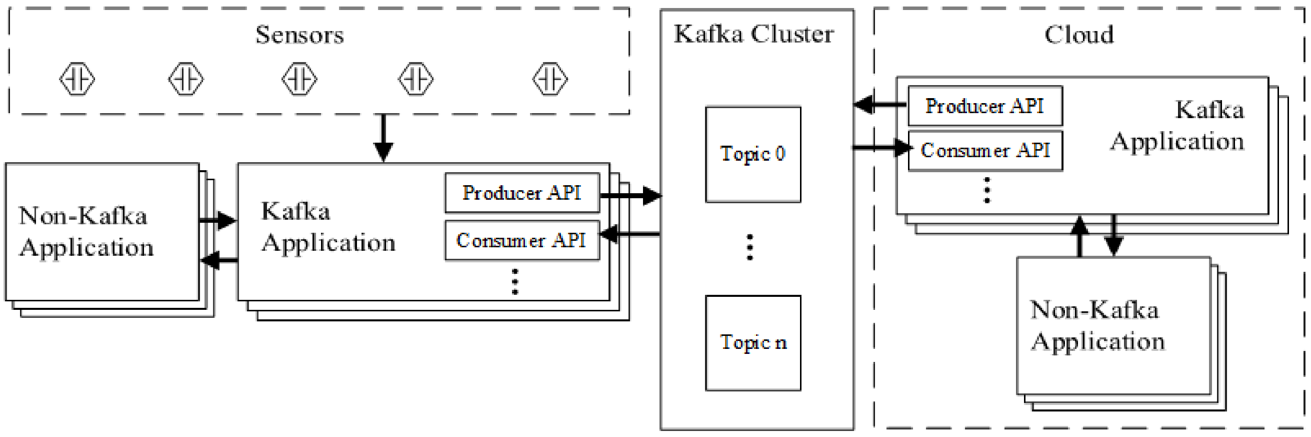

Fig. 1. General concept of using Kafka as a backbone in IoT systems.

b. *Aggregation of the grouped stream:* it involves only the data samples in which the state of the data source has changed.

c. *Embedding of aggregation data into the data stream:* we form a new message with the updated message schema that incorporates the new state change value along with timestamp parameters.

d. *State synchronization:* Kafka synchronizes the state with intermediate Kafka topics for the first operator group.

e. *Pass the result* as a sub-stream to the next operator's group.

3. *The second stage of processing* is responsible for correlation analysis between the sensor state change, received from the first operator's group, with the change of state of the valve. The following significant processing steps were implemented for this purpose:

a. *Grouping the data stream*: regroup the downstream messages of the first set of operators which involves only the detected state change for each sensor.

b. *Aggregation of the grouped stream:* aggregate the messages of the received stream in order to correlate the sensor state change and the valve state change.

c. *Embedding of aggregation data into the data stream:* we form a new message with the updated message schema that incorporates the new correlate state change value along with timestamp parameters.

d. *State synchronization:* Kafka synchronizes the state with intermediate Kafka topics for the second operator group.

e. Send the processing result message to the Result topic of the Kafka Cluster.

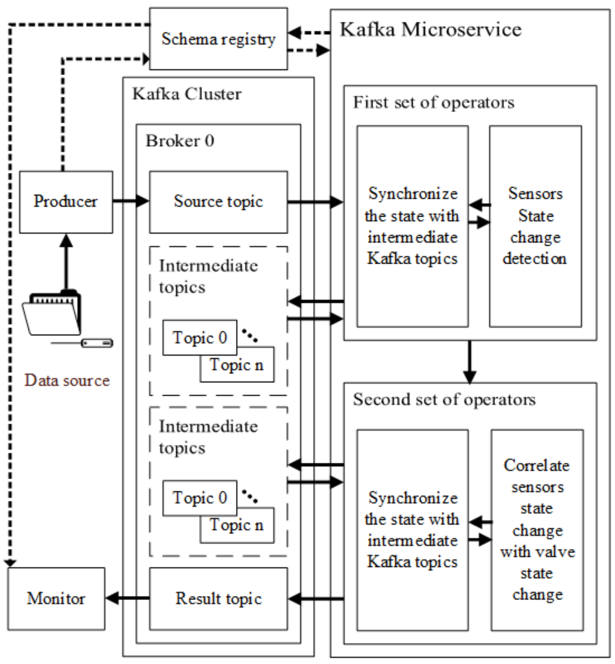

Fig. 2. Case implementation steps and its components.

V. Case-Study Implementation and Experiment Deployment

A. Tools

Docker [35] is used as a containerization platform for microservices encapsulation. When loosely coupled services interact using messages, they should provide a generic message format (schema). As a schema registry, we use the solutions, provided by the Avro [36] data serialization system, integrated into the Confluent Platform [37]. Docker image provided by Landoop [38] is used to automate the deployment of Kafka and schema registry in a container.

Kafka microservice and Producer was developed using Java 8 and Kafka Streams DSL. All data streams are serialized, exchanged, processed and deserialized as Avro data types.

In this case, a single computing node (Intel Core i7-4600U dual-core processor 2.1GHz with 16GB of RAM) was used as infrastructure support for the experiment.

B. Performance Evaluation

To measure the performance of the proposed system, we include the following additional fields to the produced messages:

- *mX_orgts*: timestamp, that indicates the moment when the sensor sends the source message number X.
- *mX_rcvts:* timestamp, that indicates the moment when the microservice receives the source message X from Kafka source topic.
- *K_sentts:* timestamp, that indicates the moment when the processed message is sent to the second set of operators.

To evaluate the efficiency of data exchange and processing, we propose to measure the following characteristics:

- *Av_SM (average interval between source messages)*: the average time delay between two consecutive source data messages sent by the producer to Kafka.
- *Av_TAT (average turnaround time)*: the average time interval required to produce a single result message by the first set of operators. Note that the interval starts from the time of receiving the second required source message (m2_rcvts), which participated in any state change from Kafka source, and ends at the time of sending the message (K_sentts) to the second set of operators.
- *Av_DV* (average delivery time): the average time interval between mX_orgts and mX_rcvts, including data serialization, acknowledgment of reception of the message in Kafka.

Results of our experiments are provided in Table I. During the one hour test, our system processed more than 379 thousand messages with the average turnaround time of about 70.7 ms.

VI. Discussion

The question of state preservation in the process of data processing is really important in tasks related to data streaming, for example, the creation of digital twins.

TABLE I. PERFORMANCE EVALUATION RESULTS

Test time	1 hour
Number of messages	379 820
Av_SM	9.5 ms
Av_TAT	70.7 ms
Av_DV	2.7 ms

Deploying data processors in containers based on cloud computing systems has a significant impact on the ability to preserve the state of such microservices. Thus, the state can be lost when a number of events occur, such as application termination in case of lack of memory or computing resources, application migration due to rebalancing, etc. In such cases, after the service restart, the local data will be empty because Kafka will restart the processing at the last committed offset in Kafka topic.

When using the Kafka Stream DSL, intermediate Kafka topics are created to store intermediate results when data transformed from one format to another. This means that Kafka Stream will create one intermediate topic for each stateful transformation. After restarting, the application scans each intermediate topic until the last committed offset. This approach does provide a high level of fault tolerance capability. On the other hand, it is implemented at the expense of increasing the overall response time.

Kafka stream DSL provides the opportunity to scale up when additional processing capacity is needed and scale down when there is no need for additional processing capacity [39]. But the number of allowed instances is limited to the number of topic partitions. We also need to consider a set of possible solutions to face the issue of scalability, considering the incrementation of the data sources needed to be processed in parallel. Further study needs to be carried out on the scalability in the situation of additional processing capacity and/or in the situation of adding a set of new data sources in the scope of the application of such approach to the Digital Twins.

VII. Conclusion

The results of our experiments have shown that the use of Kafka Stream DSL to develop stateful microservices provides a good opportunity to manage the state and an acceptable latency to support stream processing in DT. But as long as the state is stored in the Kafka cluster, the latency is affected by the amount of traffic that is created to manage data exchange with the intermediate Kafka topics to store the state. As a direction of further research, we should study and discuss the limits of the applicability of this approach. In addition, state management in Kafka Stream DSL provides a good scheme of fault tolerance in various use cases, but it is necessary to conduct further studies of fault tolerance and scalability when the problem requires strict sequential processing of data flow.

Acknowledgment

The reported study was funded by RFBR, project number 18-07-01224 A.

References

[1] P. Ferrari, A. Flammini, S. Rinaldi, E. Sisinni, D. Maffei, and M. Malara, “Impact of Quality of Service on Cloud Based Industrial IoT Applications with OPC UA,” *Electronics*, vol. 7, no. 7, p. 109, 2018.

[2] S. Tilak, P. Hubbard, M. Miller, and T. Fountain, “The Ring Buffer Network Bus (RBNB) DataTurbine Streaming Data Middleware for Environmental Observing Systems,” in *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, 2007, pp. xii–xiii.

[3] Fei Tao, Ying Zuo, Li Da Xu, and Lin Zhang, “IoT-Based Intelligent Perception and Access of Manufacturing Resource Toward Cloud Manufacturing,” *IEEE Trans. Ind. Informatics*, vol. 10, no. 2, pp. 1547–1557, May 2014.

[4] H. Kagermann, W. Wahlster, and J. Helbig, “Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Final report of the Industrie 4.0 Working Group,” *Final Rep. Ind. 4.0 WG*, no. April, p. 82, 2013.

[5] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey,” *Comput. Networks*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[6] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, “A Survey of Research on Cloud Robotics and Automation,” *IEEE Trans. Autom. Sci. Eng.*, vol. 12, no. 2, pp. 398–409, Apr. 2015.

[7] A. B. A. Alaasam, G. Radchenko, A. Tchernykh, K. Borodulin, and A. Podkorytov, “Scientific Micro-Workflows : Where Event-Driven Approach Meets Workflows to Support Digital Twins,” *Proc. Int. Conf. RuSCDays’18 - Russ. Supercomput. Days (September 24-25, 2018, Moscow, Russ. MSU*, vol. 1, pp. 489–495, 2018.

[8] M. Grieves, “Digital Twin: Manufacturing Excellence through Virtual Factory Replication,” pp. 1–7, 2014.

[9] G. Radchenko, A. Alaasam, and A. Tchernykh, “Micro-Workflows: Kafka and Kepler Fusion to Support Digital Twins of Industrial Processes,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, no. 18, pp. 83–88.

[10] J. Lee, B. Bagheri, and H. A. Kao, “A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems,” *Manuf. Lett.*, vol. 3, pp. 18–23, 2015.

[11] D. Jia, K. Lu, J. Wang, X. Zhang, and X. Shen, “A Survey on Platoon-Based Vehicular Cyber-Physical Systems,” *IEEE Commun. Surv. Tutorials*, vol. 18, no. 1, pp. 263–284, 2016.

[12] C. Stephanidis *et al.*, “Event Driven Architecture,” in *Encyclopedia of Database Systems*, Boston, MA: Springer US, 2009, pp. 1040–1044.

[13] B. B. M. Michelson and E. Links, “Event-Driven Architecture Overview,” *Patricia Seybold Gr. Res. Serv.*, 2006.

[14] D. I. Savchenko, G. I. Radchenko, and O. Taipale, “Microservices validation: Mjolnirr platform case study,” in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015 - Proceedings*, 2015, pp. 235–240.

[15] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O’Reilly Media, 2015.

[16] T. Mauro, “Adopting Microservices at Netflix: Lessons for Architectural Design,” 2015. [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/. [Accessed: 11-Jan-2019].

[17] James Lewis and Martin Fowler, “Microservices,” 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 11-Jan-2019].

[18] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Vienna: Springer Vienna, 2014.

[19] M. F. Gholami, F. Daneshgar, G. Low, and G. Beydoun, “Cloud migration process—A survey, evaluation framework, and open challenges,” *J. Syst. Softw.*, vol. 120, pp. 31–69, Oct. 2016.

[20] “Confluent: Apache Kafka as a Service, Streaming Platform for the Enterprise.” [Online]. Available: https://www.confluent.io. [Accessed: 14-Sep-2019].

[21] “Apache Kafka.” [Online]. Available: https://kafka.apache.org/. [Accessed: 31-Jan-2019].

[22] O. Carvalho, E. Roloff, and P. O. A. Navaux, “A Distributed Stream Processing based Architecture for IoT Smart Grids Monitoring,” in *Companion Proceedings of the10th International Conference on Utility and Cloud Computing - UCC ’17 Companion*, 2017, pp. 9–14.

[23] A. Sunderrajan, H. Aydt, and A. Knoll, “DEBS Grand Challenge : Real time Load Prediction and Outliers Detection using STORM,” *DEBS ’14 Proc. 8th ACM Int. Conf. Distrib. Event-Based Syst.*, pp. 294–297, 2014.

[24] “Redis.” [Online]. Available: https://redis.io/. [Accessed: 01-Feb-2019].

[25] “Apache Storm.” [Online]. Available: https://storm.apache.org/. [Accessed: 14-Sep-2019].

[26] S. Trilles, S. Schade, Ó. Belmonte, and J. Huerta, “Real-Time Anomaly Detection from Environmental Data Streams,” in *Lecture Notes in Geoinformation and Cartography*, vol. 217, 2015, pp. 125–144.

[27] “Apache ActiveMQ ™ -- Index.” [Online]. Available: http://activemq.apache.org/. [Accessed: 01-Feb-2019].

[28] A. Antonic, K. Pripuzic, M. Marjanovic, P. Skocir, G. Jezic, and I. P. Zarko, “A high throughput processing engine for taxi-generated data streams,” *Proc. 9th ACM Int. Conf. Distrib. Event-Based Syst. - DEBS ’15*, pp. 309–315, 2015.

[29] “FAQ | Apache Spark.” [Online]. Available: https://spark.apache.org/faq.html. [Accessed: 12-Sep-2019].

[30] C. Peiffer and I. L’Heureux, “(12) United States Patent,” US8346848B2, 2013.

[31] Confluent.io, “Confluent for Microservices A streaming platform based on Apache Kafka.” [Online]. Available: https://www.confluent.io/solutions/microservices/. [Accessed: 08-Jan-2019].

[32] N. Dragoni *et al.*, “Microservices: Yesterday, Today, and Tomorrow,” *Present Ulterior Softw. Eng.*, pp. 195–216, 2017.

[33] N. Alshuqayran, N. Ali, and R. Evans, “A Systematic Mapping Study in Microservice Architecture,” in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 2016, pp. 44–51.

[34] “DEBS 2012 Grand Challenge: Manufacturing equipment – DEBS.org.” [Online]. Available: http://debs.org/debs-2012-grand-challenge-manufacturing-equipment/. [Accessed: 09-Apr-2018].

[35] “Docker - Build, Ship, and Run Any App, Anywhere.” [Online]. Available: https://www.docker.com/. [Accessed: 01-Aug-2018].

[36] “Welcome to Apache Avro!” [Online]. Available: https://avro.apache.org/. [Accessed: 16-Aug-2018].

[37] Confluent .Inc, “Confluent: Apache Kafka & Streaming Platform for the Enterprise,” 2019. [Online]. Available: https://www.confluent.io/. [Accessed: 06-Sep-2019].

[38] “Landoop | Fast Data Solutions for Apache Kafka.” [Online]. Available: https://www.landoop.com/. [Accessed: 31-Jul-2018].

[39] “Elastic Scaling in the Streams API in Kafka - Confluent.” [Online]. Available: https://www.confluent.io/blog/elastic-scaling-in-kafka-streams/. [Accessed: 14-Sep-2019].