Seeking Higher Performance in Real-Time Data Processing through Complex Event Processing

Guadalupe Ortiz, Adrián Bazan-Muñoz, Pablo Caballero-Torres, Jesús Rosa-Bilbao, Inmaculada Medina-Bulo, Juan Boubeta-Puig
Department of Computer Science and Engineering
UCASE Software Engineering Group, University of Cadiz, Spain
{guadalupe.ortiz, adrian.bazan, pablo.caballero, jesus.rosa, inmaculada.medina, juan.boubeta}@uca.es

Alfonso Garcia-de-Prado
Computer Architecture and Technology Department
UCASE Software Engineering Group, University of Cadiz, Spain
e-mail: alfonso.garciadeprado@uca.es

Abstract—**Today, data processing has become a key functionality of multiple diverse applications. Large amounts of data from disparate sources must be processed in streaming in order to have real-time knowledge of the domain in question and thus be able to make the most appropriate decisions at each instant of time. This streaming processing has been successfully achieved by introducing Complex Event Processing (CEP) techniques into the solutions provided. Although these solutions have proven their effectiveness in various software architectures and application domains, there is still a need for further research on how to achieve better performance depending on the needs of the application. This paper attempts to shed some light in this area by comparing various configurations of a CEP engine, aiming for better performance in real-time data processing.**

Keywords-Complex Event Processing; Event-driven Service-oriented Architecture; Internet of Things; Data Processing.

I. INTRODUCTION

Today, data processing has become a key functionality of all applications in general and those related to the Internet of Things (IoT) and smart cities, in particular. Large amounts of data are generated from multiple sources at a high speed, which must be processed promptly to have real-time knowledge of the domain in question and thus be able to make the most appropriate decisions at each instant of time. In this context, multiple applications and architectures emerge that address big, small and open data processing, for decision making in various domains, with special emphasis on IoT and smart cities [1].

According to Rahmani et al. [2], Complex Event Processing (CEP) has become a key part of the IoT; indeed multiple publications endorse CEP as a successful technology for streaming data processing in the IoT [3]–[6], including a wide variety of works, in diverse application domains. This integration of CEP with the IoT not only takes place in the cloud, but also at levels closer to the device, such as the fog or the edge [7]. Although when we need to integrate multiple communication protocols and application technologies the use of an Enterprise Service Bus (ESB) in an event-driven service-oriented application facilitates the implementation and maintenance of the architecture [8][9]; in production environments where integration needs are lower, lighter and more efficient architectures can be achieved without using the ESB [10][11]. An architecture that integrates the CEP engine without the ESB can face with greater guarantee of success scenarios that demand higher performance, especially in the current situation where the amount and velocity of data is growing at a vertiginous rate year after year.

For all the above, we need to analyze which configurations of CEP engines can provide us with better performance in the most common scenarios of big data processing in IoT and/or smart cities; where many of the implementations are or could be limited to the integration of data sources through an inbound messaging broker with a data processing engine and an output also channeled through an outbound messaging broker. For performance analysis it is necessary to adjust to a particular implementation and given the wide use of Esper, this is going to be our CEP engine. On the other hand, given the widespread use of RabbitMQ and the immediate integration of AMQP 0.91; these are going to be the broker and protocol for both inbound and outbound messaging used in this research.

As discussed in Section III, in the past several studies on performance for CEP engines were done, such as [14][15] and [16], but we could not find particularly a comparison of 2 opposite mechanism of Esper engine to subscribe to complex events: subscriber and listener, nor the comparison of configuring CEP engines to execute with different number of threads. In this sense, this paper focused on doing the tests needed to analyze such options to check which can provide us with better performance and therefore to complement other existing research on CEP performance analysis.

The rest of the paper is organized as follows. Section II introduces CEP technology. Then, Section III explains the related work and motivates the need for further CEP testing and evaluation. Afterwards, the evaluation scenario proposed as well as the configurations of the test performed are presented in Section IV. Consequently, Section V explains the results obtained from the tests performed and, finally, Section VI presents the conclusions.

II. Background on Complex Event Processing

CEP [12] is a technology by which we can capture, analyze and correlate in real time huge amounts of data, coming from different application domains and in different formats, to detect relevant situations as they occur [13]. The incoming data to be processed by the system are called *simple events*, while the detected situations are called *complex events*.

To detect these complex events, it is necessary to have previously defined an event pattern that will be responsible for analyzing and correlating one or several simple events in a given period of time. These patterns must be deployed in a CEP engine, i.e., the software in charge of capturing the simple events, analyzing in real time if some of the patterns deployed on the simple input event stream are fulfilled, and creating the complex events.

In this work, we have adopted the Esper CEP engine and its EPL pattern language, because of its recognized prestige in terms of performance and applicability.

III. Related Work and Motivation

We have found several works which provide CEP performance evaluation. For instance, Rosa et al. [14] present a comparative study of several Esper engines for security event management. Esper CEP engine is among the engines evaluated; in their analysis we can see that Esper engine has a very good performance with a high throughput and the authors consider it to be the most suitable taking into account performance and configuration flexibility. We have also found a comparison of the Esper engine with the Sidhi CEP engine [15], in both cases integrated with an ESB and the Mosquito broker [16]. Ortiz et al. also evaluate the time it takes to transfer events in a microservice-based architecture and to process them in the Esper CEP engine [10]. Besides, Corral et al. evaluate how the integration of Esper with Kafka behaves with up to 32 partitions [17] demonstrating that the system is highly scalable under these simple conditions, but not evaluation on the CEP engine isolated, which is our main objective in this paper. Also in [11] an evaluation and comparison of Esper CEP engine in an event-driven architecture with the use of an ESB compared to the use of Data-Flows is provided, which might be complementary to the research done in this paper.

Thus, we can conclude that, to our knowledge, there is no work comparing some particular configurations of Esper CEP engine, such as the use of subscriber and listener in the engine, nor the use of several threads in its execution configuration. Such gap motivated this work which can help us to better understand Esper CEP performance and compliment other existing related works. Particularly, we expect to deploy the architecture evaluated in this paper in a water management company and we need to check which is the most efficient solution for this purpose beforehand.

IV. Evaluation Scenario

This section explains the software architecture used for the performance tests and the machines involved in it, the key performance indicators selected to be measured from the tests and the configuration prepared for the tests.

A. Architecture

The software architecture, as represented in Figure 1, consists in a synthetic data simulator (nITROGEN [18]), which submits data to a RabbitMQ broker; both deployed in *Machine 1*. The CEP application in *Machine 2* is then subscribed to the queue in the RabbitMQ broker to receive the simple events. After the simple events are processed by the CEP engine, the detected complex events are sent to an output RabbitMQ queue in *Machine 3*. The three are server machines with an Intel Xeon Silver 4110 processor and 32 GB of RAM.

B. Key Performance Indicators

To analyze in detail the processing times in each component of the architecture, we have added a series of timestamps along the life of the processed message, from its generation to the end of its processing, as explained in the following lines and shown in Figure 1.

- Let t1 be the timestamp corresponding to when the synthetic data is generated in the simulator; in this case we have used nITROGEN simulator [18].
- Let t2 be the timestamp corresponding to when the simple event (the generated synthetic data) is going to enter the CEP engine; that is, it has already been sent from the simulator to the broker and from the broker to the CEP engine.
- Let t3 be the timestamp that adds Esper CEP to the message when the complex event is detected.
- Let t4 be the timestamp corresponding to the time when the complex event leaves the CEP engine and is sent to the output queue.

Thus, the difference of t_2-t_1 indicates the time it takes for the simple event to be sent from the simulator to the messaging broker and from this to the CEP engine; that is, the sum of the sending time and the processing time in the broker. From now on we will call T_{subm} as this time difference.

On the other hand, t_3-t_2 is the time difference from the reception of the simple event in the CEP engine until the detection of the complex event in the CEP engine, i.e., the processing time of the event in the CEP engine, hereafter t_{proc}.

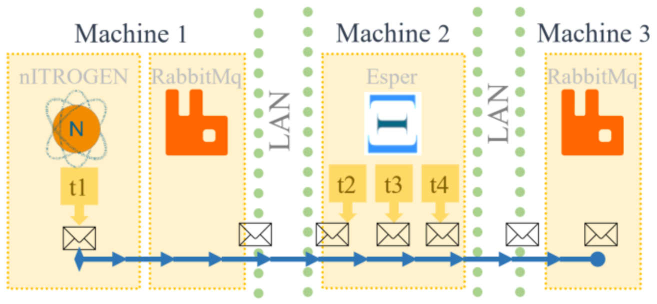


Figure 1. Software Architecture and timestamps taken.

Finally, t_4-t_2 gives us the time difference from when the simple event is going to enter the CEP engine until the complex event leaves the CEP engine to be sent to the output queue; that is, it includes not only the processing time in CEP of the simple event, but also the management of the complex event in the CEP engine. From now on this time will be called T_{man}.

Such three times (T_{subm}, T_{proc}, T_{man}) together with the CPU usage and memory consumption will be the key performance indicators in our evaluation tests.

C. Test Configuration

The objective of these tests is not to evaluate several instructions of the EPL syntax to build patterns, as we did in the past [11], but to evaluate several ways to handle the complex event and configure the CEP engine to process a simple pattern and check specifically if the use of the *listener* and the *subscriber* clauses in a pattern, as well as the configuration of the CEP engine execution using 1 or 10 threads in the processing, influence the performance of the whole architecture.

Let us explain that a *listener* can subscribe to complex events already posted by a pattern match. However, a *subscriber* object receives statement results via method invocation. A subscriber is expected to have performance advantages, but we will have to see if this holds true in the performance tests. We used a very simple event scheme and event pattern for the evaluation with the aim of insolating the behavior of the listener and the subscriber and make it independent of any pattern clause.

The simple events reaching the CEP engine will consist on a JSON element containing i) the timestamp of the instant of creation of the event (*t1*); ii) the timestamp of when it reaches the CEP engine (*t2*), which will be generated empty by default and added when such data is known and iii) a boolean —*shouldTrigger*—that will cause the pattern to be met randomly or not for each incoming event; which is represented as follows in the Esper CEP engine:

```
    @public @buseventtype create json
schema Dummy as (t1 long, t2 long,
shouldTrigger boolean)
```

The pattern will simply add the timestamp of the instant in which the complex event is detected (*t3*) and select the other timestamps that were already in the simple event (*t1* and *t2*).

```
    insert into DummyComplexEvent SELECT
current_timestamp as t3, t2, t1 FROM
Dummy(shouldTrigger)
```

As previously mentioned, the reason for using such a simple event pattern is because we want to focus on the different behavior of the system using the subscriber and listener, as well as executing with one or more threads. It is not our aim to evaluate a wide range of Esper operators as this was done by other works, but to complement such works with this novel tests.

Every test was run for simple events incoming rates of 1 000, 10 000 and up to 50 000 incoming events per second, and each test was run for 10 minutes. As previously said, T_{subm}, T_{proc}, T_{man}, CPU usage and memory usage were measured for every performed test.

V. Results and Discussion

In this section we show and analyze the results of the tests performed both with subscriber and listener and for 1 and 10 threads for the CEP engine execution under the conditions described in Section III.

As we can see in Table I and Table II, the use of one thread, either with listener or with subscriber, seems to have an average consumption of memory and CPU quite similar for any of the tested incoming rates. Also, the submission times from the message queue to the CEP engine are quite similar, as they should be. We can also see that the processing time is also almost the same in both cases, but we note some differences in the management time: even though the subscriber seems to be more efficient than the listener when we have an input rate of 1 000 events per second (0.264 ms the listener versus 0.02 ms of the subscriber) when we reach the input rate of 50 000 events per second, the listener is the one being more efficient (0.007 ms the listener versus 0.33 ms of the subscriber). It is important to point out that the high values reached for T*man* with the input rate of 50 000 events per second are due to the fact that the system collapses and therefore does not process all the messages properly and may give inconsistent values.

Again, as we can see in Table III and Table IV, average consumption of memory and CPU are also quite similar when using ten threads with any of the tested incoming rates and the submission times from the message queue to the CEP engine are quite similar, as well. In this occasion we can see that the processing time is again similar for both the subscriber and listener options. This time, the management time for the listener behaves better (0.14 versus 0.69 ms) at a 10 000 events per second incoming rate, as well as the rate of 50 000 incoming events per second (0.78 ms of the listener versus 0.99 ms of the subscriber).

TABLE I. Test Results with Listener Configuration and 1 Thread.

Incoming Rate (events/s)	Medium Memory Usage (MB)	Medium CPU Usage (%)	T_{subm} (ms)	T_{proc} (ms)	T_{man} (ms)
1 000	466.5	0.37	10.77	0.007	0.264
10 000	518.2	1.35	58.05	0.005	0.58
50 000	520.3	3.38	4 853	0.0034	0.07

TABLE II. Test Results with Subscriber Configuration and 1 Thread.

Incoming Rate (events/s)	Medium Memory Usage (MB)	Medium CPU Usage (%)	T_{subm} (ms)	T_{proc} (ms)	T_{man} (ms)
1 000	463.3	0.34	10.78	0.007	0.02
10 000	535.8	1.35	63.70	0.005	0.86
50 000	524.4	3.53	4 387	0.0038	0.33

TABLE III. TEST RESULTS WITH LISTENER CONFIGURATION AND 10 THREADS.

Incoming Rate (events/s)	Medium Memory Usage (MB)	Medium CPU Usage (%)	T_{subm} (ms)	T_{proc} (ms)	T_{man} (ms)
1 000	485.8	0.67	9.30	0.89	0.86
10 000	517.3	3.62	58.26	0.14	7.66
50 000	7 884.8	20.51	257.18	0.78	85 114.69

TABLE IV. TEST RESULTS WITH SUBSCRIBER CONFIGURATION AND 10 THREADS.

Incoming Rate (events/s)	Medium Memory Usage (MB)	Medium CPU Usage (%)	T_{subm} (ms)	T_{proc} (ms)	T_{man} (ms)
1 000	483.7	0.65	7.90	0.7	0.9
10 000	516.1	2.86	58.36	0.69	4.63
50 000	8 089.6	19.71	264.31	0.99	87 522

Up to this point of the comparison we can say that for simple events there are no big differences between using a listener or a subscriber because although there are some differences at some rates of incoming events per second, they are not significant, not reaching the millisecond.

There are differences between the use of 1 or 10 threads in the execution of the CEP engine, although perhaps not the expected ones. To better observe these differences, we have represented in Figure 2 three graphs with the values taken by T_{subm}, T_{proc} and T_{man}, respectively, for each input rate with the listener and the subscriber and the execution in 1 thread; and these same three graphs but using 10 threads for the execution in Figure 3.

For the time of submission (t_{subm}) we do not appreciate big differences (as expected). However, for the time of processing in the CEP engine (t_{proc}), when using a single thread, the processing time increases as the input rate of simple events increases; however, the processing time decreases when using 10 threads (until it collapses at 50 incoming events per second). On the other hand, if we take as a reference the input rate 10 000 events per second, in which the engine is not collapsed but it is not as fluid as with 1 000 input events, we see that we obtain better times with 1 thread than with 10; possibly due to the greater management involved in the distribution of tasks among the threads and the resolution of the final results. Finally, the processing and management time (t_{man}) increases in both cases as we increase the input rate of simple events, until it saturates at 50000 input events per second; but it remains in any case lower for the execution with 1 thread compared to the one using 10 threads.

VI. THREADS TO VALIDITY

A limited number of tests have been performed in this work. As previously mentioned, a single pattern has been tested, but to better validate the results, perhaps a varied set of operators or domain specific patterns could be tested. On the other hand, generating a greater or lesser number of complex events for each simple input event may yield other results. It should also be noted that the use of 1 thread has been compared with the use of 10 threads, but other intermediate options such as 2, 3, 4, etc. threads have not been tested. Tests with different numbers of threads could lead to other conclusions in addition to those explained in this paper.

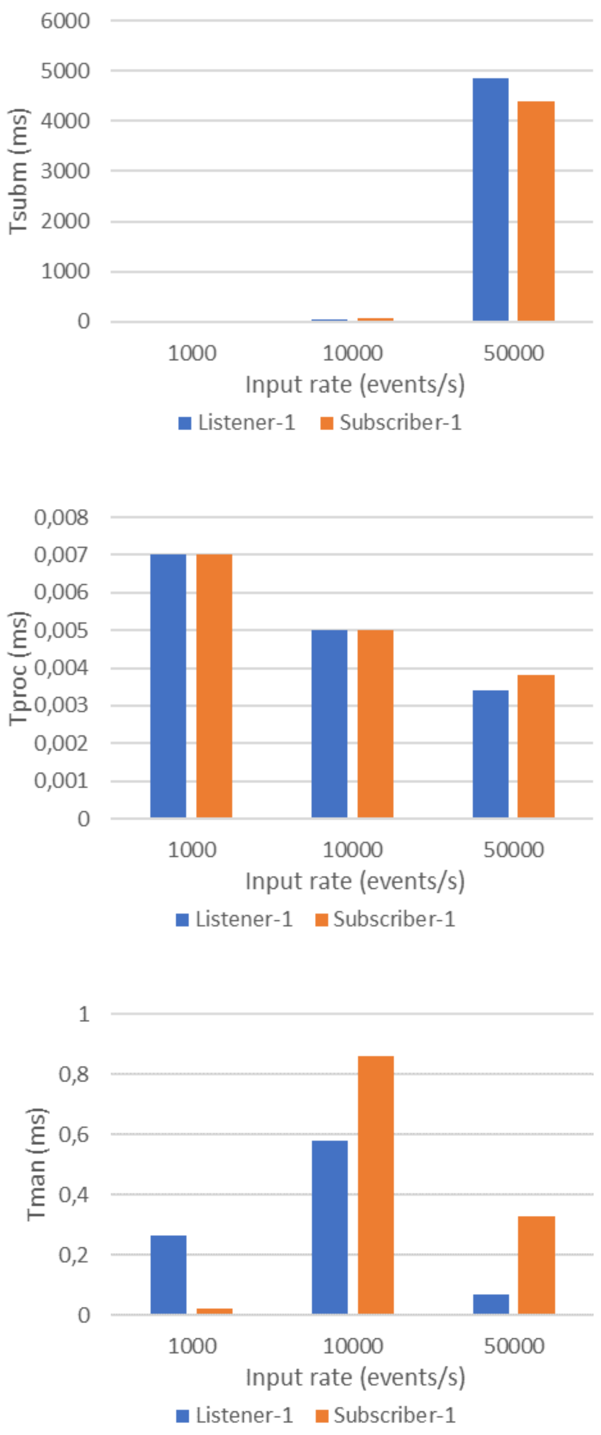
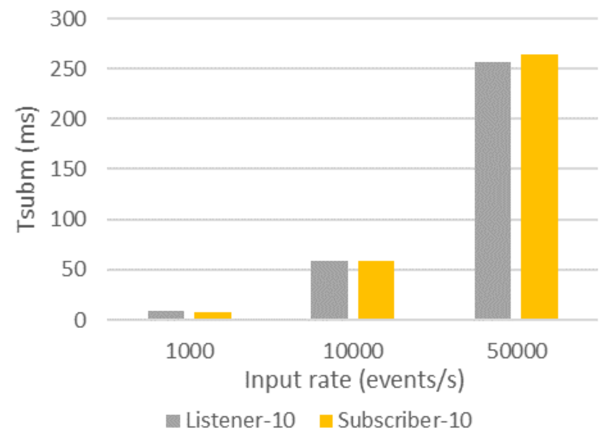
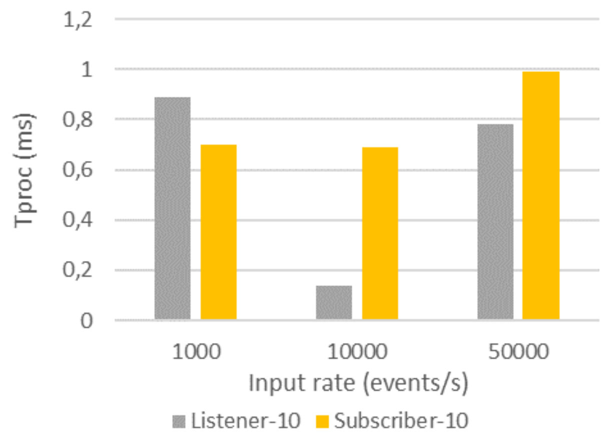


Figure 2. T_{subm}, T_{proc} and T_{man} for each input rate with the listener and the subscriber and the execution in 1 thread.

On the other hand, it is important to bear in mind that in real systems, when measuring processing time, a lack of synchronisation in the clocks of the systems involved may imply a mismatch in the measurement of these times. However, when implementing a real system, our main goal will not be to measure performance time, but rather for the patterns to detect the situations of interest in the domain in question. In this case, clock times have no influence, since the events are processed the instant they arrive, regardless of the timestamp they may contain, and it is the CEP engine that assigns the timestamps necessary for the internal management of the times and time windows.





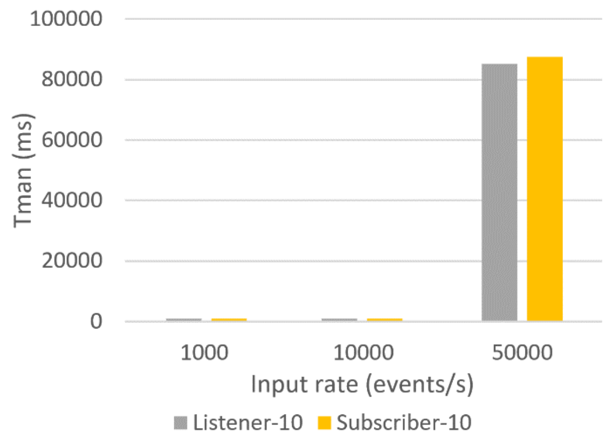


Figure 3. Tsubm, Tproc and Tman for each input rate with the listener and the subscriber and the execution in 1 thread.

VII. CONCLUSION AND FUTURE WORK

In light of the results of the tests performed, we can conclude that both the use of subscriber and listener to capture the complex events detected by the CEP engine provide similar behaviour at different rates of incoming events per second. We can also conclude that configuring the CEP engine to use more threads might not be useful when we have large amounts of incoming events as it is more time consuming to distribute and assign the tasks for the different threads than the time required to do it in a single thread.

For future work, we expect to perform further performance tests with the particular patterns developed for the water management company where we will test the architecture evaluated in this paper.

ACKNOWLEDGMENT

This work was partly supported by grant PDC2022-133522-I00 (ASSENTER project) funded by MCIN/AEI /10.13039/501100011033 and by the" European Union Next GenerationEU/ PRTR" and partly by the grant program for R&D&i projects, for universities and public research entities qualified as agents of the Andalusian Knowledge System, within the scope of the Andalusian Plan for Research, Development and Innovation (PAIDI 2020). Project 80% co-financed by the European Union, within the framework of the Andalusia ERDF Operational Program 2014-2020 "Smart growth: an economy based on knowledge and innovation". Project funded by the Ministry of Economic Transformation, Industry, Knowledge and Universities of the Andalusian Regional Government. DECISION project with reference P20_00865. We are also grateful for the collaboration of the water supply network management company GEN.

REFERENCES

[1] S. D. Liang, "Smart and Fast Data Processing for Deep Learning in Internet of Things: Less is More", *IEEE Internet Things J.*, vol. 6, no. 4, pp. 5981–5989, Aug. 2019, doi: 10.1109/JIOT.2018.2864579.

[2] A. M. Rahmani, Z. Babaei, and A. Souri, "Event-driven IoT architecture for data analysis of reliable healthcare application using complex event processing", *Clust. Comput.*, vol. 24, no. 2, pp. 1347–1360, Jun. 2021, doi: 10.1007/s10586-020-03189-w.

[3] A. Akbar, A. Khan, F. Carrez, and K. Moessner, "Predictive Analytics for Complex IoT Data Streams", *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1571–1582, Oct. 2017, doi: 10.1109/JIOT.2017.2712672.

[4] R. Mayer, B. Koldehofe, and K. Rothermel, "Predictable Low-Latency Event Detection With Parallel Complex Event Processing", *IEEE Internet Things J.*, vol. 2, no. 4, pp. 274–286, Aug. 2015, doi: 10.1109/JIOT.2015.2397316.

[5] A. Garcia-de-Prado, G. Ortiz, and J. Boubeta-Puig, "CARED-SOA: A Context-Aware Event-Driven Service-Oriented Architecture", *IEEE Access*, vol. 5, pp. 4646–4663, 2017, doi: 10.1109/ACCESS.2017.2679338.

[6] A. Garcia-de-Prado, G. Ortiz, and J. Boubeta-Puig, "COLLECT: COLLaborativE ConText-aware service oriented architecture for intelligent decision-making in the Internet of Things", *Expert Syst. Appl.*, vol. 85, pp. 231–248, Nov. 2017, doi: 10.1016/j.eswa.2017.05.034.

[7] G. Mondragón-Ruiz, A. Tenorio-Trigoso, M. Castillo-Cara, B. Caminero, and C. Carrión, "An experimental study of fog and cloud computing in CEP-based Real-Time IoT applications", *J. Cloud Comput.*, vol. 10, no. 1, p. 32, Dec. 2021, doi: 10.1186/s13677-021-00245-7.

[8] H. Derhamy, J. Eliasson, and J. Delsing, "IoT Interoperability—On-Demand and Low Latency Transparent Multiprotocol Translator", *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1754–1763, Oct. 2017, doi: 10.1109/JIOT.2017.2697718.

[9] A. Massaro *et al.*, "Production Optimization Monitoring System Implementing Artificial Intelligence and Big Data", in *2020 IEEE International Workshop on Metrology for Industry 4.0 & IoT*, Roma, Italy, Jun. 2020, pp. 570–575. doi: 10.1109/MetroInd4.0IoT48571.2020.9138198.

[10] G. Ortiz *et al.*, ‘A microservice architecture for real-time IoT data processing: A reusable Web of things approach for smart ports’, *Comput. Stand. Interfaces*, vol. 81, p. 103604, Apr. 2022, doi: 10.1016/j.csi.2021.103604.

[11] G. Ortiz, I. Castillo, A. Garcia-de-Prado, and J. Boubeta-Puig, "Evaluating a Flow-based Programming Approach as an Alternative for Developing CEP Applications in the IoT", *IEEE Internet Things J.*, vol. 9, no. 13, pp. 11489–11499, 2021, doi: 10.1109/JIOT.2021.3130498.

[12] D. C. Luckham, *Event processing for business: organizing the real-time enterprise*. Hoboken, N.J, USA: John Wiley & Sons, 2012.

[13] C. Inzinger, W. Hummer, B. Satzger, P. Leitner, and S. Dustdar, "Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems: event-based monitoring and adaptation for distributed systems", *Softw. Pract. Exp.*, vol. 44, no. 7, pp. 805–822, Jul. 2014, doi: 10.1002/spe.2254.

[14] L. Rosa, P. G. Alves, T. J. Cruz, and P. Simoes, "A Comparative Study of Correlation Engines for Security Event Management", presented at the Int. Conf. on Cyber Warfare and Security, Kruger National Park, South Africa., 2015.

[15] WSO2, "Siddhi", 2022. http://siddhi.io/ (accessed Mar. 20, 2023).

[16] J. Roldán, J. Boubeta-Puig, J. Luis Martínez, and G. Ortiz, "Integrating complex event processing and machine learning: An intelligent architecture for detecting IoT security attacks", *Expert Syst. Appl.*, vol. 149, p. 113251, Jul. 2020, doi: 10.1016/j.eswa.2020.113251.

[17] D. Corral-Plaza, G. Ortiz, I. Medina-Bulo, and J. Boubeta-Puig, "MEdit4CEP-SP: A model-driven solution to improve decision-making through user-friendly management and real-time processing of heterogeneous data streams", *Knowl.-Based Syst.*, vol. 213, p. 106682, Feb. 2021, doi: 10.1016/j.knosys.2020.106682.

[18] A. Garcia-de-Prado, "nITROGEN: Internet of Things RandOm GENreator", 2020. https://ucase.uca.es/nITROGEN/ (accessed Mar. 20, 2023).