Computers and Electrical Engineering 92 (2021) 107110

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Simplifying IoT data stream enrichment and analytics in the edge ☆

Márcio Miguel Gomes [a], Rodrigo da Rosa Righi [a,*], Cristiano André da Costa [a], Dalvan Griebler [b]

[a] Applied Computing Graduate Program - Unisinos University, Rio Grande do Sul, Brazil
[b] School of Technology - Pontifical Catholic University of Rio Grande do Sul, Brazil

ARTICLE INFO

Keywords:
Edge computing
IoT
Stream processing
Data analysis
Framework

ABSTRACT

Edge devices are usually limited in resources. They often send data to the cloud, where techniques such as filtering, aggregation, classification, pattern detection, and prediction are performed. This process results in critical issues such as data loss, high response time, and overhead. On the other hand, processing data in the edge is not a simple task due to devices' heterogeneity, resource limitations, a variety of programming languages and standards. In this context, this work proposes STEAM, a framework for developing data stream processing applications in the edge targeting hardware-limited devices. As the main contribution, STEAM enables the development of applications for different platforms, with standardized functions and class structures that use consolidated IoT data formats and communication protocols. Moreover, the experiments revealed the viability of stream processing in the edge resulting in the reduction of response time without compromising the quality of results.

1. Introduction

In recent decades, technological advances in computing – related to device miniaturization and increased computing power as well as connectivity and software solutions – have resulted in the spread of "smart" devices for a variety of purposes. These advances have enabled the development of the "Internet of Things", or simply "IoT", a landscape where "smart" devices are equipped with a variety of sensors that can read the environment in which they operate, have microcontrollers or microprocessors to perform calculations, logic, and algorithms to communicate autonomously with each other or with cloud-hosted systems. Typical passive appliances such as dishwashers, air conditioners, coffee makers, televisions and even vehicles today have data collection, processing and network communication capabilities. We also have wearable devices, which monitor, process and transmit various vital signals such as heart rate, body temperature, blood pressure and step count. Therefore, technological evolution resulted in a huge increase in data collection, processing, and transmission.

To perform the processing of data streams in real-time, David Luckham developed the concept of "Complex Event Processing", or simply CEP [1]. He proposed using a set of techniques such as rule-based systems, pattern detection, event correlation and time window over streams of raw data or simple events to build a system capable of inferring an emerging pattern over a set of diverse data sources in real-time. This makes it possible not only to identify a complex event in real-time, but also to anticipate an event or even identify that an event should have happened, but did not happen. The first implementation of CEP middleware was Rapide, a project developed in Stanford University. After Rapide, several initiatives were launched, as Apache Flink, Drools, Esper and

☆ This paper was submitted for regular issues, but it is for special section VSI-eiia. Reviews processed and recommended for publication by Guest Editor Dr. Jiafu Wan.

* Corresponding author.
E-mail address: rrrighi@unisinos.br (R. da Rosa Righi).

https://doi.org/10.1016/j.compeleceng.2021.107110
Received 1 July 2020; Received in revised form 18 February 2021; Accepted 12 March 2021
Available online 26 March 2021
0045-7906/

Siddhi. Although CEP is consolidating as a standard technology for real-time data processing in IoT environment, it is centralized and requires significant computational resources such as CPU, memory, network, database, etc [2].

Due to limited computational resources, IoT devices usually rely on cloud-hosted stream processing centralized platforms to perform complex analytical processing like filtering, aggregation, classification, pattern detection, and prediction. This remote data processing results in critical issues such as connection loss, high response time, and overhead in the centralized computing system [2]. But performing data processing on the network edge is not a simple task. IoT devices are heterogeneous in hardware architecture and limited in processor, memory and communication capabilities. The development of custom-made embedded software, the variety of technologies and programming languages besides lacking standards is not a simple problem to solve.

Analyzing the literature, we found several initiatives to bring data analytics in real-time to the network edge, discussed in detail in Section 3. The most recent works had as main objectives the reduction of both network traffic [3,4] and network latency [5–7] and as secondary objectives the reduction of memory consumption [8], reduction of energy usage [4], data classification [5,9] and recovery of incomplete data series [10]. The most used techniques were pattern recognition [3,4,7], outlier detection [4–7], and prediction [5–7]. A common characteristic of all works is that they are handcrafted solutions, implemented from scratch, not using a standard development framework or API (Application Programming Interface).

Our motivation for developing the current work is the lacking of standardization in IoT application development, the heterogeneity of IoT hardware and data formats, the variety of communication protocols between edge and cloud applications, and the complexity in implementing analytic functions in the fog. Aiming to standardize and simplify the development of IoT applications, we are proposing STEAM, an architecture to bring data processing functions from cloud to the network edge, reducing response time, and allowing decision-making locally. In addition, the outcome of data analytics is merged with original data using a technique known as stream enrichment. Thus, client applications, middlewares, and stream processors hosted in the cloud or local network can consume data streams enriched with valuable data.

The contributions of the article are threefold:

- An IoT model that enables us to connect different communication protocols, enrich data, abstract devices and implement data acquisition.
- A programming framework that simplifies the development of IoT data stream enrichment and analytics in the edge.
- An experimental evaluation that demonstrates the feasibility of the IoT model and programming framework under a real-world scenario.

The rest of the paper is structured as follows. Section 2 presents the background of concepts and technologies that are the basis of our work. Section 3 is a summary of research and work related to stream processing in the edge. Our proposed model is discussed in Section 4, followed by the detailing of the evaluation methodology in Section 5. The results are presented and discussed in Section 6 and Section 7 concludes the article.

2. Background

This section presents an overview of basic concepts and technologies involved on the development of the STEAM model and framework.

2.1. Fog and mist computing

In fog computing, computation is performed by gateway devices at the network edge, reducing bandwidth requirements, latency, and the need for communicating data to the servers [11]. However, in a strict definition of fog computing, the devices at the edge are not involved in computation but only in data acquisition, while the interpretation occurs in the gateway. Thus, network delay and inefficient bandwidth utilization are still present. On the other hand, mist computing pushes processing even further to the network edge, residing directly within the network fabric, bringing the fog computing layer closer to the smart end-devices such as sensors and actuators [12]. This approach decreases latency and increases the autonomy of the subsystems since the calculation and performance are performed locally and depend mainly on the device's perception of the situation.

2.2. Real-time systems

The term "real-time" means the requirement for IT systems to process data as they arrive rather than storing the data and retrieving it at some point in the future. The time interval for computation and response is typically in order of milli, micro, or even nanoseconds [13]. Input data usually arrives from various data sources, in different formats and protocols. Outputs often involve generating notifications to humans about significant occurrences in the environment or invoking functions of the system's components to perform some tasks. Real-time systems must meet the following requirements: low latency, high availability and horizontal scalability [14].

Table 1
Related work and their main features.

Reference	Year	Proposal	Objective		Technique		
			Traffic reduction	Latency reduction	Prediction	Pattern recognition	Outlier detection
Khairnar et al. [5]	2020	Middleware		✓	✓		✓
Galanopoulos et al. [3]	2020	Algorithm	✓	✓		✓	
Ali et al. [6]	2019	Framework		✓	✓		✓
Babazadeh [4]	2019	Architecture	✓			✓	✓
Symeonides et al. [7]	2019	Framework		✓	✓	✓	✓
Bharath Das et al. [17]	2018	Framework		✓	✓	✓	
Harth et al. [18]	2018	Architecture	✓		✓		
Lujic et al. [10]	2018	Algorithm		✓	✓		✓
Oyekanlu et al. [19]	2018	Algorithm		✓			
Alam et al. [20]	2017	Framework	✓	✓		✓	✓
Harth and Anagnostopoulos [21]	2017	Architecture	✓	✓	✓	✓	✓
Portelli and Anagnostopoulos [22]	2017	Algorithm	✓		✓		
Tsai et al. [23]	2017	Middleware	✓	✓		✓	
Bhargava et al. [8]	2016	Algorithm				✓	✓
Kartakis et al. [24]	2016	Algorithm	✓	✓		✓	✓

2.3. Stream processing

Many real-time applications work over a continuous data flow, which is received, processed and dispatched indefinitely. This type of interaction is called a stream and it implies the transfer of a sequence of related information, like raw data or simple events [13]. The main difference between them is that while data streams work with raw data, event streams work with previously processed and categorized events [15]. Raw data are independent and have no specific meaning, but events include an identifier, event attributes, and the moment of occurrence.

2.4. Complex event processing

Some applications need to extract more valuable insights from data performing complex computations over multiple streams, coming from the same or different sources by combining or correlating events. These relationships can be of any type, including causal relationships, data relationships, or temporal dependencies. Simple events can be consolidated into complex events through several transformations such as threshold-based filtering, joining, sequencing, and well-known aggregation functions [16]. This kind of event processing is often referred to as CEP [1].

3. Related work

With the objective to get an overview of IoT data processing at the network edge, we made a bibliographic research selecting articles from relevant conferences and journals. The research was conducted in April 2020, searching for works published since the year 2016, aiming to identify articles containing the keywords “Event Processing” and “Edge”, which are the domains of most interest. It was also necessary for the articles to present at least one of the terms “analytics”, “prediction”, “pattern” or “outlier”, which are the most specific subjects. The outcome is listed in Table 1. In the sequence, we present a discussion of the main related works.

A middleware for monitoring industrial processes using IoT analytics was proposed by [5]. The work consisted of reading temperature, vibration, and humidity as well as training an ANN (Artificial Neural Network) for prediction of machinery malfunction. A dynamic and distributed algorithm was developed by [3] with the objective of improving the execution of data analytics at IoT devices, with more robust instances running at cloudlets. They used KNN (K-Nearest Neighbors) and CNN (Convolutional Neural Network) techniques for image recognition and classification.

A framework for historical data analysis combined with real-time data analytics and forecasting was proposed in [6]. The authors used Multiple Linear Regression, Support Vector Regression, Decision Tree Regression and Random Forest Regression. In [7] the authors proposed a declarative query model aiming to abstract the complexity of defining data analytics rules. They proposed a grammar and a framework tailored for edge computing to achieve latency, robustness and even privacy requirements.

A distributed architecture was proposed in [4] to detect infrastructure anomalies, using an adaptation of Lempel–Ziv–Oberhumer (LZO) data compression library, Kalman filter and thresholds. Authors made experiments using a Wireless Sensor Network (WSN). In [17] authors built a framework for efficient distribution of stream processing tasks among multiple edge devices, based on the proximity of sensors (data sources) and computational capacity of devices. They used *Seagull*, an extension of *Cowbird* framework.

In order to reduce network traffic, [18,21] proposed a lightweight distributed architecture executing on the edge, resulting in predictive intelligence. They used Exponentially Weighted Moving Average (EWMA), Multivariate Linear Regression (MLR) and Reconstruction Vectors (RV) techniques. With the same objective of reducing network traffic, the work presented in [9] implemented algorithms to calculate kurtosis, asymmetry, mean square root and crest factors, analyzing mechanical vibration data in industrial machines. Only abnormal situations were reported to a cloud-based system.

In [10] authors proposed a set of algorithms for recovering incomplete time-series data sets at the network edge. They used ARIMA (Auto-regressive Integrated Moving Average), EWMA, and TBATS (Trigonometric Exponential Smoothing State Space model with Box–Cox transformation, ARMA errors, Trend and Seasonal Components), reducing prediction errors and execution time. An experimental study was conducted in [20] to evaluate the performance of 21 Frequent Pattern Mining (FPM) algorithms over data streams. The author used far-edge computing and observed memory consumption and execution time for comparing purpose. Identifying water leaking was the objective of [24], combining a lightweight algorithm executed on the edge to detect anomalies based on data compression rates and geolocation using graph theory. Authors also used Moving Average (MA), Kalman filter and thresholds. Experiments resulted in a highly precise physical location of water leaking and reduction in data communication.

The work developed in [22] aimed to obtain a highly accurate prediction using regression over data streams. The authors used the techniques of Linear Regression (LR) and Adaptive Vector Quantization (AVQ). Only metadata of the models was transmitted from edge to cloud. In [8] authors proposed compressing data in a Wireless Sensor Network (WSN) besides using L-SIP (Linear Spanish Inquisition Protocol) to offer real-time feedbacks allowing accurate data reconstruction. They converted raw data in state vectors, storing only instances that could not be predicted.

Analyzing the related works, we found several efforts to bring stream analytics from the cloud to the network edge, resulting in a reduction of latency and network traffic. However, the initiatives were tailored to specific situations, lacking a common architecture or standardization. The edge devices most used in the experiments were notebooks, smartphones or single-board computers, such as Raspberry Pi, and only one experiment used Arduino LoRa. We identified a set of common data analysis techniques to process pattern recognition, outlier detection, and prediction, highlighting the desired features of an IoT application. We also perceived a tendency of decreasing the importance of cloud processing for real-time applications. The remote systems are associated with supervisory and management tasks, such as processing high-level business rules, event detection and reporting, logging, and data visualization.

4. STEAM model

In this section, we present STEAM, a model and framework designed to enable real-time data analytics and data streams enrichment at the network edge. STEAM is an acronym for *Stream Enrichment and Analysis in the Mist*. The design guidelines are presented in Section 4.1, the architecture is detailed in Section 4.2, while data processing is explored in Section 4.3.

4.1. Design guidelines

The IoT environment is well-known for its limited computational resources and heterogeneity in data formats, protocols, and hardware architectures. On the other hand, cloud-hosted applications rely on standard protocols for capturing data besides applying consolidated analytic functions. Therefore, these characteristics guided us in defining the STEAM model and framework, providing flexibility and simplicity in developing an IoT application. To deal with limited computational resources and heterogeneity of hardware architectures, we propose a device abstraction layer, allowing a standard and lightweight access to IoT devices. To manage the variety of data formats and protocols, we designed a data acquisition layer, allowing us to work with the most varied data formats and acquisition protocols from the sensor network. For data processing, we built an extensible function library that provides several consolidated techniques such as pattern recognition, clustering, prediction, among others. And finally, a protocol connector interfaces with cloud services throughout consolidated protocols, such as HTTP, MQTT, and CoAP, but not limited to them.

4.2. Architecture

Fig. 1 represents a high-level overview of our architecture. On the left side (a), we can see the standard architecture usually adopted in IoT applications. On the right side (b), we can see the STEAM architecture for comparison purposes and have a better understanding of our contribution.

A typical IoT application begins with data production, represented as generic raw data sources transmitted over sensor networks. Data can be produced by both sensors and simple applications, embedded in dedicated hardware. After collected, raw data are processed by a gateway at the network edge, which usually only encapsulates the data frames in a standard protocol and transmits to client applications using Intranet or Internet. Since we propose to bring data analytics techniques to the network edge applying the mist computing concept, we highlight the *Analysis* and *Enrichment* processes executed on far-edge devices. Lastly, the client applications are responsible for data consumption and business rules processing and can be hosted either on LAN or cloud.

Fig. 2 depicts a detailed view of STEAM model. It consists of a four-layered framework for the development of applications targeting resource-limited devices located at the network edge. The input layer is *Device Abstraction and Data Collection*, responsible for capturing data from sensors and far-edge devices, standardizing and forwarding data streams to the processing step. The processing step is composed of 2 layers. The first one is *Analysis*, that provides a set of data analysis techniques, such as filtering, transformation, pattern recognition, outlier detection, prediction, etc. The second processing layer is *Enrichment*, intended to merge the outcome of the previously mentioned *Analysis* layer along with the original data streams. The output layer is the *Protocol Connector*, responsible for providing output data streams in a standard format and different communication protocols, so that client applications can have access to data in a standard and transparent manner. The following describes each layer in detail.

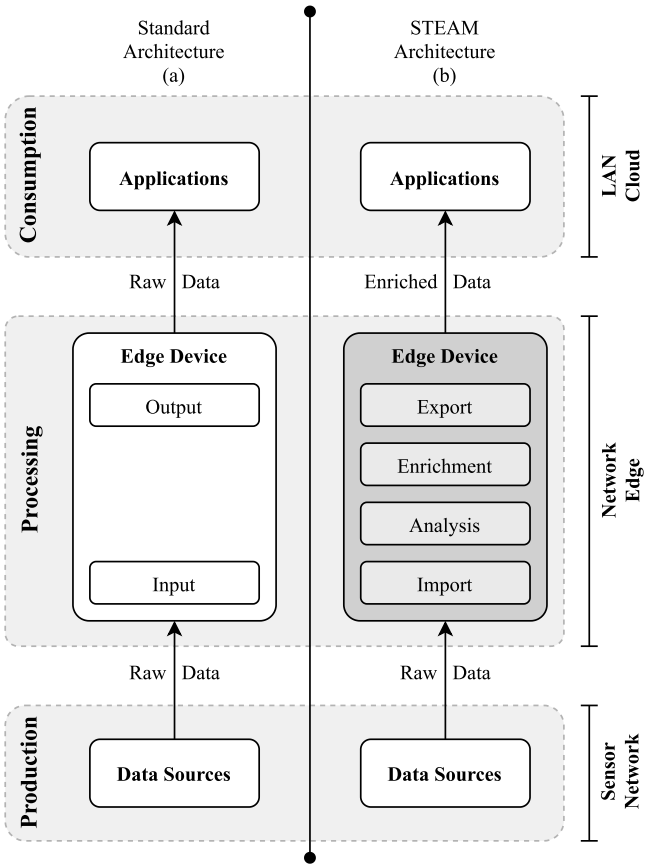

Fig. 1. Overview of Standard IoT Architecture (a) compared to STEAM Architecture (b). In (a), raw data is captured from sensors and sent to applications through far-edge devices, that acts only as a proxy between production and consumption layers. However, in (b), far-edge devices are more than a proxy, they are responsible for processing and analyzing data in real-time, sending enriched data streams to middlewares or final applications.

4.3. Data processing

There are several steps between reading raw data from a sensor until the detection of a complex event. Next, we thoroughly present how STEAM is designed for processing data captured from sensors to providing enriched streams to cloud-hosted applications.

i. **Device Abstraction and Data Acquisition**: Since data sources can contain different formats and use different communication protocols, the *Device Abstraction and Data Communication* layer aims to provide transparent interfacing with sensors and devices. For this, it uses either open or proprietary protocols, such as RS-232, TCP, UDP, HTTP, among others to capture data in various formats, such as ASCII, JSON, XML and even binary. Once data is received, it is parsed, extracted and organized in a standard format, then, it is sent to the *Analysis* layer. Standard format consists of a JSON containing the attributes "id", "value", "unit" and "timestamp". In Fig. 3, we have an example of a raw data frame in ASCII format transmitted via RS-232 serial communication representing a temperature measurement of 23.6 Celsius degrees, converted to a standard JSON format.
ii. **Analysis**: The *Analysis* layer consists of a set of modules that perform the most varied operations on the received data in order to extract relevant information. Initially, we propose 6 modules that can be executed independently or combined in sequence:

- **Preprocessing**: This module if responsible for eliminating null or invalid values, discard values out of a predefined range, normalize or convert values from one measurement unit to another, etc;
- **Aggregation**: In this step, we apply aggregation functions to a data set, such as min, max, sum, count, etc. We are using either temporal and batch sliding windows over the data set to select a subset of values;
- **Statistics**: This module calculates statistics over a data set, like average, median, standard deviation, variance, skewness, and kurtosis. We also can select a subset of values to be processed, using temporal or batch sliding windows;

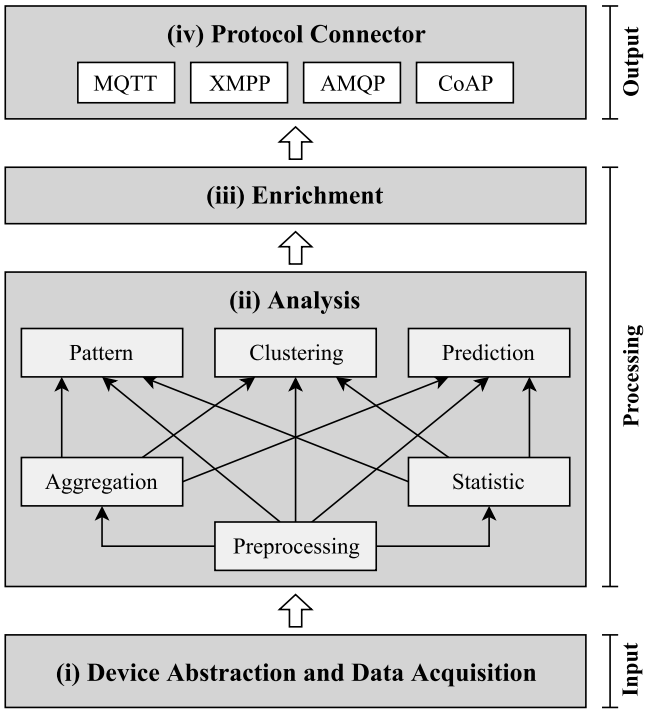

Fig. 2. Detailed STEAM model. The input occurs in *Device Abstraction and Data Acquisition Layer*, which standardizes and sends raw data to the *Processing* layer. The *Analysis* step consists of several modules that can be executed independently or in sequence, resulting in one or more calculated values. These values are sent to the *Enrichment* layer and merged with original raw data. The output consists of the *Protocol Connector* layer, which transmits the data using consolidated IoT protocols.

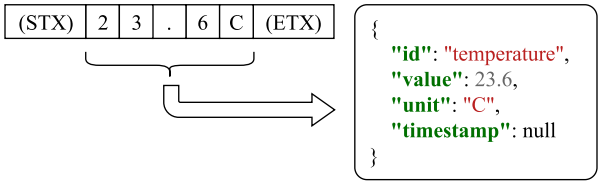

Fig. 3. Data received in ASCII format and converted to JSON.

- **Pattern**: This module applies trend and seasonality analysis to identify behavior patterns, such as stability, upward or downward trend and their respective rates, trend changing, etc. We make this applying a smoothing technique, such as MA or EWMA and auto-correlation analysis;
- **Clustering**: This module consists of partitioning data sets into groups based on similarity or distance so that data in the same cluster are similar regarding an arbitrary attribute. First, we need to define an appropriate distance/similarity metric, and then, use a clustering technique, such as k-means, hierarchical clustering, density-based clustering or subspace clustering over an attribute.
- **Prediction**: After identifying the past behavior pattern, this module predicts a future value at one or several steps ahead of the current time, along with a reliability factor. Here we use techniques such as ARIMA, TBATS, Kalman filter and ANN.

iii. **Enrichment**: The *Enrichment* layer consists of adding the result of the analysis step into the original data packet before it is transmitted to the applications. Since in the related work no data standardization initiative was identified, we are proposing a list of identifiers with suggestive names and their respective data types. The names used for the output attributes can be viewed in Table 2. Besides attribute names and data types, we are proposing a JSON structure for data formatting and standardization, and the schema is depicted in Fig. 4.

iv. **Protocol Connector**: The output layer is the *Protocol Connector*. Because it is responsible for sending to client applications the data provided by the *Enrichment* layer, we are using consolidated IoT protocols to provide maximum inter-connectivity between the Sensor Network and LAN/Cloud layers. According to studies conducted by [25,26], the most commonly used protocols in IoT environment are AMQP, CoAP, DDS, MQTT, RESTFUL Services, WAMP and XMPP.

Table 2
Standardization of nomenclatures used by the enrichment module.

Attribute	Required	Type	Description
id	Yes	Int/text	Data packet or data source id
value	Yes	Any	Data value
unit	No	Text	Measurement unit
timestamp	No	Datetime	Instant of data acquisition
min	No	Number	Min value
max	No	Number	Max value
sum	No	Number	Sum of values
count	No	Number	Count of values
average	No	Number	Arithmetic average
ewaverage	No	Number	Exponentially weighted average
median	No	Number	Median
variance	No	Number	Variance
stdeviation	No	Number	Standard deviation
slope	No	Number	Trending slope
outlier	No	Boolean	Value out of standard
filtered	No	Boolean	Value out of thresholds

```
{
 "type": "object",
 "properties": {
  "id":        {"type": ["integer", "string"]},
  "value":     {"type": "any"},
  "unit":      {"type": ["string", "null"]},
  "timestamp": {"type": ["string", "null"],
                "format": "date-time"}
 },
 "required": ["id", "value"],
 "additionalProperties": {
  "properties": {
   "min":          {"type": "number"},
   "max":          {"type": "number"},
   "sum":          {"type": "number"},
   "count":        {"type": "number"},
   "average":      {"type": "number"},
   "ewaverage":    {"type": "number"},
   "median":       {"type": "number"},
   "variance":     {"type": "number"},
   "stdeviation": {"type": "number"},
   "skewness":     {"type": "number"},
   "kurtosis":     {"type": "number"},
   "slope":        {"type": "number"},
   "outlier":      {"type": "boolean"},
   "filtered":     {"type": "boolean"}
  }
 }
}
```

Fig. 4. JSON schema for data formatting and standardization.

5. Evaluation methodology

In order to prove the viability of the STEAM model, we developed a framework that provides all functionalities previously discussed in Section 4. The design of the framework prototype is presented in Section 5.1. The infrastructure for testing the framework is shown in Section 5.2, and the evaluation metrics are presented in Section 5.3. With the framework, we developed one application targeting limited computational resources devices, described in Section 5.4. Finally, the test scenario is detailed in Section 5.5.

5.1. Framework prototype

Fig. 5 depicts the STEAM framework class diagram. The main class is *Device*, which is the base class for creating an application. It is a virtual class, an interface that defines a set of attributes and methods that must be implemented by sub-classes. The *Device*'s sub-classes represent the *Device Abstraction and Data Acquisition* layer. They are intended to implement the communication methods between the data sources and the STEAM application. As a proof-of-concept, we implemented *Analog* and *Serial* (RS-232) classes. The *Device* class also uses a *Parser* class, responsible for parsing and transforming the incoming raw data collected from sensors in a standard output format. The virtual class *Function* represents the *Analysis* layer in STEAM model, and defines the interface for analytic functions, such as filtering, prediction, outlier detection, and more, which must be implemented as sub-classes of *Function*.

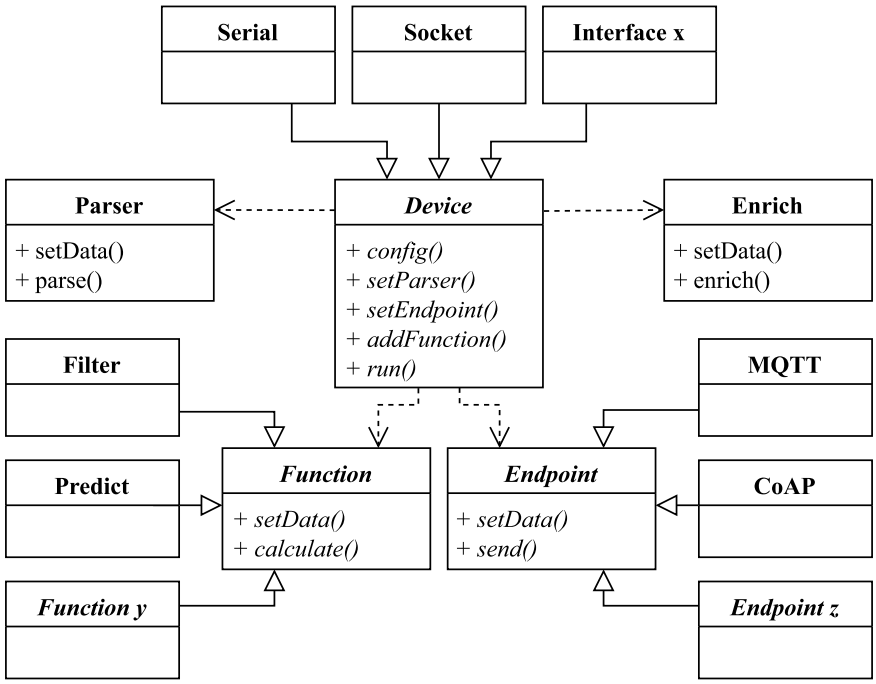

Fig. 5. STEAM Framework class diagram.

A *Device* object can reference one or more *Function* objects, that are stored in a list structure and are executed as a flow, from the first to the last function, to achieve the desired data processing.

The *Enrich* class represents the *Enrichment* layer, and is responsible for merging the outcome of *Functions* execution flow with the original data. One or more attributes with the results are included in the data packet, which is converted to a standard format and sent to the last processing step. The output of STEAM framework consists in sending the data packet previously processed using a standard IoT protocol. The *Protocol Connector* layer is implemented by *Endpoint* class, which defines the interface for data communication protocols with client applications running on the local network or cloud. For this prototype, *Endpoint* sub-classes implement consolidated IoT protocols, such as MQTT, CoAP, AMQP and HTTP RESTful.

5.2. Infrastructure

The environment set up for the experiment of assessing STEAM application presents the following architecture, depicted in Fig. 6. We have one sensor that constantly sends its measured value via RS-232 serial communication to the STEAM application embedded on an IoT device, that consists of one Raspberry Pi 2 running Raspbian OS. The *Device Abstraction and Data Acquisition* layer of the STEAM application continuously receives raw data packets sent from the sensor through RS-232 serial communication and encapsulates the data in the previously mentioned standard format. After, the raw data stream is sent to *Analysis* layer and the outcome is merged with the original data stream in the *Enrichment* layer. Finally, the *Protocol Connector* layer of STEAM application sends the enriched stream to a CEP Engine hosted on Azure. A wireless access point is responsible for linking the IoT device network to the cloud.

The CEP Engine is *WSO2 Streaming Integrator* running *Siddhi* and was installed in a VM hosted in Azure running Ubuntu 18.04. The CEP Engine receives data as an HTTP stream in JSON format coming from STEAM application, parses and processes the data, and sends the result back to IoT device asynchronously.

5.3. Evaluation metrics

For the quality assessment of STEAM, we are using three metrics: *Accuracy*, *Time Consumption* and *Data Packet Size*. *Accuracy* is measured comparing the values computed on the edge by STEAM application against the values returned by CEP Engine computed on cloud. Since one of the objectives of this work is to bring data analytics from cloud to edge, the outcomes of both processing should be the same. To measure accuracy according to Fig. 6, we uniquely identify *(UID)* each data packet processed on the edge *(EP)* and send them *(Snd)* to the CEP Engine for cloud processing *(CP)*. Once *CP* is complete, we send back an asynchronous response *(Res)* packet to the edge identified with the same *UID*. Then, we match the *EP* and *CP* packages by *UID* and compare the results. This way, the accuracy metric consists of counting mismatching values comparing the *EP* and *CP* packages during the entire test.

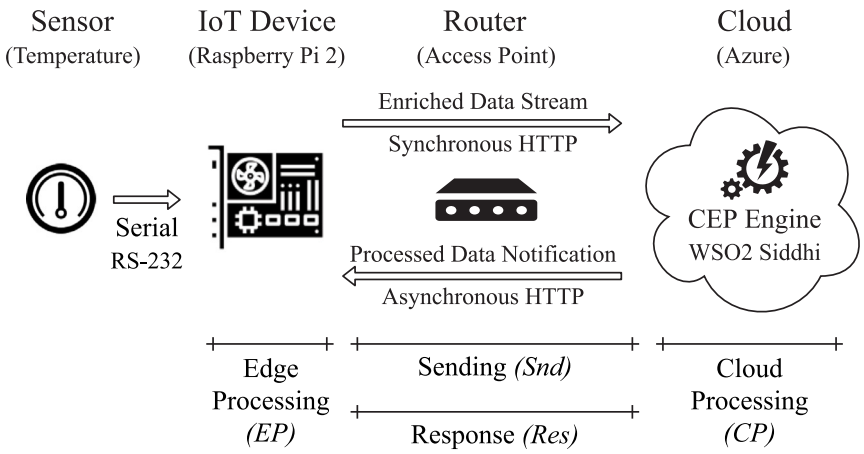

Fig. 6. Infrastructure for evaluating a STEAM application and comparing with a hybrid edge/cloud application.

Regarding *Time Consumption*, we expect a significant decrease comparing an exclusive edge processing *(EP)* application versus a cloud processing *(CP)* dependent application. Although a STEAM application demands more processing on the edge (EP), and consequently spends more time locally processing than a hybrid edge/cloud application, we eliminate data transmission *(Snd + Res)*, resulting in a shorter time interval for obtaining the same final result, besides the complete elimination on timeouts and packages loss. This way, the *Time Consumption* metric for a STEAM application accounts only *EP* time, depicted in Eq. (1), while in a hybrid edge/cloud application it is computed by the sum of *EP*, *Snd*, *CP* and *Res* times, as defined in Eq. (2).

$$Edge = EP \tag{1}$$

$$Hybrid = EP + Snd + CP + Res \tag{2}$$

At last, the *Data Packet Size* metric registers the amount of data generated by the *Enrichment* process and compares it against a raw data packet. The increase of data packet size directly impacts network traffic and can be a problem according to the protocol used between the edge and the cloud. Therefore, we are analyzing not only the total increase in data size but how many bytes each enriched data on average injects into the packet.

5.4. Application

The STEAM application was developed by following a sequence of object instantiating and message exchanging depicted in Fig. 7. When the application starts, it calls the *config()* method of the *Device* object. The configuration varies depending on the type of communication. For example, to work with analog signals we specify the port where the sensor is attached, sampling frequency and the ADC max supported voltage. For serial RS-232, we specify the port, speed, parity, data bits, and stop bits parameters. Following, we define the *Parser* object that will be used for interpreting and standardizing raw data by calling the *setParser()* method. Next, the *Endpoint* object is defined by calling the *setEndpoint()* method. The *Endpoint* object will later send data to the client application using standard IoT protocols such as MQTT, CoAP or AMQP. The *addFunction()* method adds one or more *Function* objects to the application. Each *Function* object is responsible for applying a distinct analytical function to the input data, generating a result that will be included later in the data packet by an *Enrich* object.

When we execute the *run()* method, the application passes the execution control to the *Device* object, which is responsible for executing the application in an infinite loop. This loop starts with the execution of the *readData()* method, which waits for data to arrive from communication with the sensor network. As soon as a data packet is received by the device, it calls the *setData()* method of the *Parser* object, and then receives the data in a standard format by calling the *parse()* function. Standardized data is passed to one or more *Function* objects using the *setData()* method. The *calculate()* function is fired for each *Function* object, which processes the data and returns the processed result. The results of the analytic functions are sent to the *Enrich* object via the *setData()* method, and returned by the *enrich()* function. At this point, the enriched data packet is sent to the *Endpoint()* object via the *setData()* method, and is finally sent to the client application via the *send()* method. This finishes the processing cycle of a data packet and the loop returns to the initial instruction, awaiting the arrival of a new data packet to repeat the entire processing sequence again.

5.5. Scenario

To assess the STEAM framework, we created one application for monitoring the temperature inside a clean-room of a microchip manufacturer. We are using a temperature sensor that constantly sends its value via RS-232 serial communication to the STEAM application running in a Raspberry Pi 2 with Raspbian OS, at a transmission rate of 20 measurements per second. The data stream

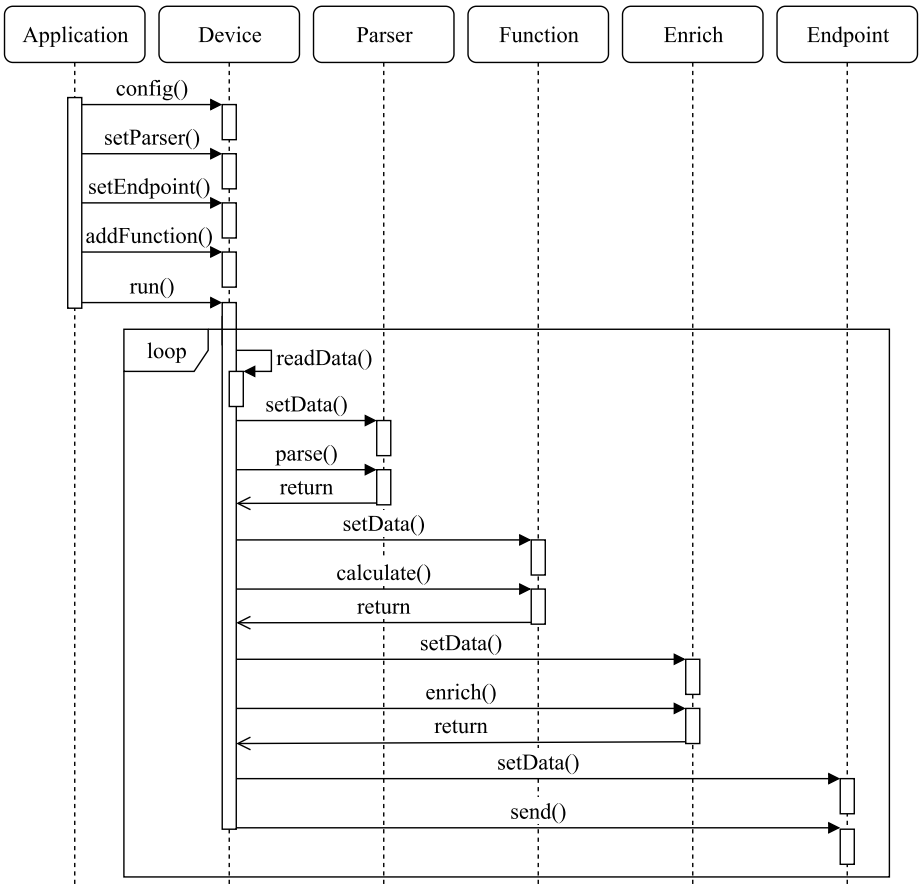

Fig. 7. Sequence diagram of a STEAM application.

```
-- Input stream, receiving data from edge
@source(type = 'http', basic.auth.enabled = "false", @map(type = 'json',
  receiver.url = "http://0.0.0.0:8006/inputStream"))
define stream InputStream (id long, value double, unit string, timestamp long);

-- Aggregate and calculation functions over window length = 20
@info(name='qryAggregate')
from InputStream#window.length(20)
select
  id, count() as countVal, sum(value) as sumVal, min(value) as minVal,
  max(value) as maxVal, avg(value) as avgVal, stdDev(value) as stdDevVal,
  ifThenElse(stdDev(value) > 1.0, true, false) as outlier
insert into AggregateStream;

-- Sends the outcome computed in the cloud back to the edge
@sink(type = 'http', method = "POST", @map(type = 'json',
  publisher.url = "http://[STEAM_IoT_machine_IP]/wso2"))
define stream AggregateStream (id long, countVal long, sumVal double,
  minVal double, maxVal double, avgVal double, stdDevVal double, outlier bool);
```

Fig. 8. Application configured in WSO2 Streaming Integrator hosted in a VM in Azure. First, we define the input stream that receives data through an HTTP post, then a query executes the aggregation functions and finally, the data is sent back to the edge.

also is sent to CEP Engine in the cloud, which is running the application depicted in Fig. 8, makes the same calculation over the same sliding window and asynchronously sends the results back to the edge.

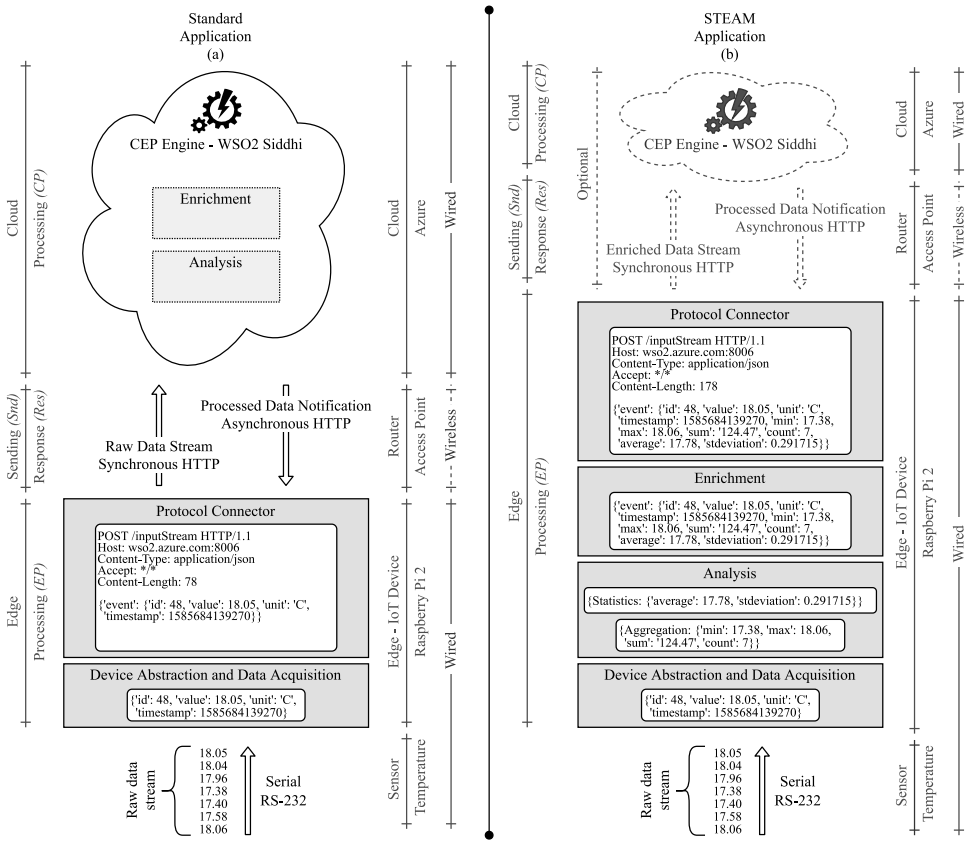

Fig. 9. STEAM application in detail, starting from data acquisition, analysis, enrichment, and sending to a CEP Engine cloud-hosted. In a Standard Application (a), all data analysis is performed in the Cloud, while in a STEAM Application (b), the computation is performed in the Edge, making the Cloud an optional element.

We performed 30 tests, each sending 530 measures to the applications through a data stream. The applications calculate the count, sum, minimum, maximum, average, standard deviation and outlier indicator over a sliding window of 20 values. We are not processing the first 19 data packets since, at the beginning of the test, the sliding window still does not have 20 events. From the 20th event to the end, each new received data fires a new window, then, we perform the calculation. This way, we are processing data 510 times per test, 20 times per second, each test lasting approximately 25.5 s.

6. Results and discussion

To have a better understanding of the entire data flow and transformation, we monitored each step of data processing in the STEAM application, depicted in Fig. 9, detailed in (a) *Standard Application*, acting as a proxy between the sensor and the CEP Engine, and (b) *STEAM Application*, performing analysis and enrichment in the network edge.

In this diagram, the *Device Abstraction and Data Acquisition* layer captures raw data from the sensor and converts it to a standard format, in both scenarios. While in (a) we readily send formatted data to the cloud via *Protocol Connector*, where the data stream is transmitted synchronously to the CEP Engine in the cloud using the HTTP protocol, in (b) we perform several processing steps. The *Analysis* layer computes statistics and aggregation over the data stream, then, we send the outcome to the *Enrichment* layer, which merges the processed data with original data, and finally sends it to the *Protocol Connector*. For both scenarios, when the CEP Engine ends the computation, it sends back a notification to the STEAM application using an asynchronous HTTP request.

In both scenarios, (a) and (b), the CEP Engine hosted in the cloud is responsible for data analytics and computation, then sending the result back to the edge. However, in (b), the STEAM application makes the same computation and obtain the same result, making the CEP Engine an optional element.

According to Section 5.3, the defined metrics to evaluate the STEAM framework were *accuracy*, *time consumption* and *data packet size*. Regarding accuracy, we compared the result of calculations made in the edge against the values returned by the cloud, event by event, and we found no mismatches. Thus, we achieved an accuracy of 100% in calculations. However, we have identified some packet loss. From 30 tests with 530 events each summarizing 15 900 data frames, only 8 did not return to STEAM application. Analyzing the logs in detail, we identified that all data have arrived in the CEP engine and were correctly processed, but somehow,

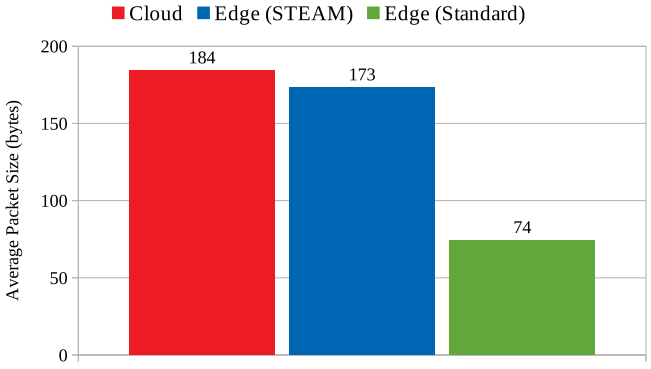

Fig. 10. Average packet size in bytes. Edge (Standard) packets include only basic data while Edge (STEAM) and Cloud include enriched data.

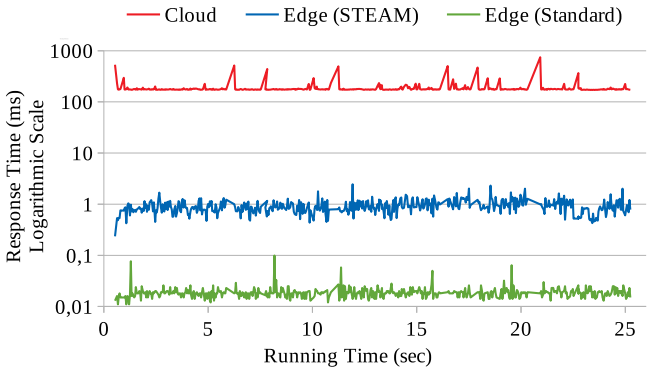

Fig. 11. Response time of one single test that represents the common behavior of all performed tests. The *y*-axis is in logarithmic scale.

they got lost on the way back to the edge. It is worth investigating the reasons for this problem in a future work, but they are outside of the current research scope.

Considering data packet size in JSON format, we identified the increase of bytes amount after the enrichment process, illustrated in Fig. 10. A single raw data packet with the attributes *id*, *value*, *unit* and *timestamp* add up to 74 bytes. The STEAM enrichment process in the edge for six attributes *min*, *max*, *sum*, *count*, *average*, and *stdeviation* injected nearly 99 bytes into the packet, resulting in 173 bytes per packet on average. Finally, these six attributes computed in the cloud resulted in a 184 bytes packet, increasing the original raw data packet in 110 bytes. The average packets size increasing due to the enrichment process is significant, reaching 133.8% for the STEAM application in the edge and 148.6% for the CEP application in the cloud. Even though the increase in the size of the data packets has been significant, sending the data to the cloud has become an optional task, since the values are now calculated by the STEAM application in the edge.

Regarding time consumption, Fig. 11 depicts one single test that exemplifies the common behavior of all 30 accomplished tests. The red line at the top of the chart is the processing time in scenario (a) for every data, computed by Eq. (2). The green line at the bottom of the chart represents only the time consumed in the edge by a standard application, according to Eq. (1). The blue line at the center of the chart is the processing time spent by STEAM application in scenario (b), running on the edge, using the same Eq. (1). The *y*-axis represents the response time in milliseconds, expressed on a logarithmic scale due to the large difference in the magnitudes involved in both edge and cloud response times. The *x*-axis is the test running time, expressed in seconds.

The values presented in Table 3 are the minimum, maximum, average, median and standard deviation of times involved in the experiments. The *Hybrid* column represents the whole scenario depicted in Fig. 9(a), computing $EP + Snd + CP + Res$ times, while the column *Standard* is considering only EP time over the same scenario. The *STEAM* column also shows the EP time, however, over Fig. 9(b) scenario. The average response time in all tests involving all requests made to the cloud was 185 ms with a standard deviation of 45.3323. The minimum response time was 171 ms and the maximum was 747 ms. Among all requests made to the CEP Engine, 95% lasted from 171 ms to 223 ms, while 85% spent between 171 ms and 180 ms. Analyzing the time used in the internal processing of the STEAM application running on a Raspberry Pi 2, we obtained an average of 910 µs per processed data frame, with a standard deviation of 275.5413. The faster processing time was 236 µs, and the slowest was 2449 µs. For 95% of packets processed, it took between 425 µs to 1361 µs, while 70% of processing happened from 670 µs to 1200 µs. Uniquely comparing the average response time, the STEAM application was 230.3 times faster than a hybrid edge-cloud application, with the same accuracy in calculations and discarding the dependency of a complex cloud environment setup.

Table 3
Processing and response times from the experiments.

Metric/Scenario	Hybrid edge + cloud	Standard edge	STEAM edge
Minimum	171 ms	11 µs	236 µs
Maximum	747 ms	99 µs	2449 µs
Average	185 ms	18.88 µs	910 µs
Median	176 ms	18 µs	854 µs
Standard deviation	45.3323	6.1562	275.54138

Table 4
Qualitative comparison between literature and STEAM.

Characteristic	Literature	STEAM
Application	Specific	General
Development	Hard coded	Class API
Analytics	Hard coded	Function library
Data acquisition	Specific	Device abstraction
Data export	Specific	Protocol connector

We also compared the STEAM application against the literature presented in Section 3. We could not make quantitative analysis since related work did not provide details about the experiments as well as the source of data sets. Therefore, we made a qualitative comparison presented in Table 4. The applications developed in the related works are aimed at specific purposes and require hard-coded development not only for the software structure but data input, output, analytics, and logic. However, the STEAM applications are for general purposes, built over a standard class API, and helped by an analytics function library. Furthermore, in STEAM applications, data acquisition and export are managed respectively by the *Device Abstraction* and *Protocol Connector* layers, providing abstraction and extensibility seamlessly.

7. Conclusion

Although IoT technologies has been researched and improved for the last two decades, the lacking of standardization in application development is still a current problem. In this work we proposed STEAM, an architecture to bring data processing functions from cloud to the network edge, reducing response time and allowing decision-making locally. Besides this, we proposed the enrichment of data streams with locally computed values, using a standard format and nomenclatures, offering valuable data to middlewares, CEP Engines or custom-made applications hosted in the cloud.

To prove the viability of STEAM, we implemented a framework and developed one application for capturing data from a temperature sensor, computing values and transmitting both a raw and a enriched data stream to a CEP Engine hosted in Azure. As a result, we obtained a drastic reduction in response time, elimination of timeouts and packets loss without giving up the precision of the outcomes. In this restricted context, a STEAM application brought data analytics from the cloud to the network edge, making the need of a CEP Engine unnecessary, allowing energy savings, reduction of computational structure and, consequently, of financial resources.

Until now, we focused our efforts on validating concepts and implementing features related to infrastructure. In future work, we intend to develop more analytic modules, such as prediction and pattern recognition. We also plan to implement the STEAM framework in C++ language, targeting devices with limited computing resources, like ATMega and ESP micro-controllers.

CRediT authorship contribution statement

Márcio Miguel Gomes: Writing, Reading, Revised this version of the manuscript. **Rodrigo da Rosa Righi:** Writing, Reading, Revised this version of the manuscript. **Cristiano André da Costa:** Writing, Reading, Revised this version of the manuscript. **Dalvan Griebler:** Writing, Reading, Revised this version of the manuscript.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the following Brazilian agencies: CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) and FAPERGS (Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul).

References

[1] Luckham David C. The power of events: An introduction to complex event processing in distributed enterprise systems. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 2001.

[2] Yasumoto Keiichi, Yamaguchi Hirozumi, Shigeno Hiroshi. Survey of real-time processing technologies of IoT data streams. J Inf Process 2016;24(2):195–202. http://dx.doi.org/10.2197/ipsjjip.24.195.

[3] Galanopoulos Apostolos, Tasiopoulos Argyrios G, Iosifidis George, Salonidis Theodoros, Leith Douglas J. Improving iot analytics through selective edge execution. 2020, arXiv preprint arXiv:2003.03588.

[4] Babazadeh Mehrdad. Edge analytics for anomaly detection in water networks by an arduino101-lora based WSN. ISA Trans 2019. http://dx.doi.org/10.1016/j.isatra.2019.01.015.

[5] Khairnar Vaishali, Kolhe Likhesh, Bhagat Sudhanshu, Sahu Ronak, Kumar Ankit, Shaikh Sohail. Industrial automation of process for transformer monitoring system using IoT analytics. In: Inventive communication and computational technologies. Springer; 2020, p. 1191–200.

[6] Ali Muhammad Intizar, Patel Pankesh, Breslin John G. Middleware for real-time event detection andpredictive analytics in smart manufacturing. In: 2019 15th international conference on distributed computing in sensor systems (DCOSS). IEEE; 2019, p. 370–6. http://dx.doi.org/10.1109/DCOSS.2019.00079.

[7] Symeonides Moysis, Trihinas Demetris, Georgiou Zacharias, Pallis George, Dikaiakos Marios. Query-driven descriptive analytics for IoT and edge computing. In: 2019 IEEE international conference on cloud engineering (IC2E). IEEE; 2019, p. 1–11. http://dx.doi.org/10.1109/IC2E.2019.00-12.

[8] Bhargava Kriti, Ivanov Stepan, Donnelly William, Kulatunga Chamil. Using edge analytics to improve data collection in precision dairy farming. In: 2016 IEEE 41st conference on local computer networks workshops (LCN workshops). IEEE; 2016, p. 137–44. http://dx.doi.org/10.1109/LCN.2016.039.

[9] Oyekanlu Emmanuel. Predictive edge computing for time series of industrial IoT and large scale critical infrastructure based on open-source software analytic of big data. In: 2017 IEEE international conference on big data (Big Data), Vol. 2018-Janua. IEEE; 2017, p. 1663–9. http://dx.doi.org/10.1109/BigData.2017.8258103.

[10] Lujic I, De Maio V, Brandic I. Adaptive recovery of incomplete datasets for edge analytics. In: 2018 IEEE 2nd international conference on fog and edge computing (ICFEC). 2018, p. 1–10. http://dx.doi.org/10.1109/CFEC.2018.8358726.

[11] Preden JS, Tammemäe K, Jantsch A, Leier M, Riid A, Calis E. The benefits of self-awareness and attention in fog and mist computing. Computer 2015;48(7):37–45. http://dx.doi.org/10.1109/MC.2015.207.

[12] Iorga Michaela, Feldman Larry, Barton Robert, Martin Michael J, Goren Ned, Mahmoudi Charif. Fog computing conceptual model. Technical report, Gaithersburg, MD: National Institute of Standards and Technology; 2018, p. 11. http://dx.doi.org/10.6028/NIST.SP.500-325.

[13] Buyya Rajkumar, Calheiros Rodrigo N, Dastjerdi Amir Vahid. Big data: principles and paradigms. Morgan Kaufmann; 2016.

[14] Ellis Byron. Real-time analytics: Techniques to analyze and visualize streaming data. John Wiley & Sons; 2014.

[15] Teng Piyun, Li Guanyu, Su Lei, Chen Xinying. Adaptive rule update method in complex event process. In: Proceedings - 9th international conference on intelligent human-machine systems and cybernetics, IHMSC 2017, Vol. 1. Institute of Electrical and Electronics Engineers Inc.; 2017, p. 270–5. http://dx.doi.org/10.1109/IHMSC.2017.69.

[16] Incki Koray, Ari Ismail, Sozer Hasan. Runtime verification of IoT systems using complex event processing. In: Vasilakos A V, Zhou M, Lukszo Z, Palau C, Liotta A, Vinci A, Basile F, Fanti M P, Guerrieri A, Fortino G, editors. 2017 IEEE 14th international conference on networking, sensing and control (ICNSC). IEEE; 2017, p. 625–30. http://dx.doi.org/10.1109/ICNSC.2017.8000163, URL http://ieeexplore.ieee.org/document/8000163/.

[17] Bharath Das Roshan, Di Bernardo Gabriele, Bal Henri. Large scale stream analytics using a resource-constrained edge. In: 2018 IEEE international conference on edge computing (EDGE). IEEE; 2018, p. 135–9. http://dx.doi.org/10.1109/EDGE.2018.00027.

[18] Harth Natascha, Anagnostopoulos Christos, Pezaros Dimitrios. Predictive intelligence to the edge: impact on edge analytics. Evol Syst 2018;9(2):95–118. http://dx.doi.org/10.1007/s12530-017-9190-z.

[19] Oyekanlu Emmanuel, Onidare Samuel, Oladele Paul. Towards statistical machine learning for edge analytics in large scale networks: Real-time Gaussian function generation with generic DSP. In: 2018 first international colloquium on smart grid metrology (SmaGriMet). IEEE; 2018, p. 1–6. http://dx.doi.org/10.23919/SMAGRIMET.2018.8369850.

[20] Alam Khubaib Amjad, Ahmad Rodina, Ko Kwangman. Enabling far-edge analytics: Performance profiling of frequent pattern mining algorithms. IEEE Access 2017;5:8236–49. http://dx.doi.org/10.1109/ACCESS.2017.2699172.

[21] Harth Natascha, Anagnostopoulos Christos. Quality-aware aggregation and predictive analytics at the edge. In: 2017 IEEE international conference on big data (Big Data), Vol. 2018-Janua. IEEE; 2017, p. 17–26. http://dx.doi.org/10.1109/BigData.2017.8257907.

[22] Portelli Kurt, Anagnostopoulos Christos. Leveraging edge computing through collaborative machine learning. In: 2017 5th international conference on future internet of things and cloud workshops (FiCloudW), Vol. 2017-Janua. IEEE; 2017, p. 164–9. http://dx.doi.org/10.1109/FiCloudW.2017.72.

[23] Tsai Pei-Hsuan, Hong Hua-Jun, Cheng An-Chieh, Hsu Cheng-Hsin. Distributed analytics in fog computing platforms using tensorflow and kubernetes. In: 2017 19th asia-pacific network operations and management symposium (APNOMS). IEEE; 2017, p. 145–50. http://dx.doi.org/10.1109/APNOMS.2017.8094194.

[24] Kartakis Sokratis, Yu Weiren, Akhavan Reza, McCann Julie A. Adaptive edge analytics for distributed networked control of water systems. In: 2016 IEEE first international conference on internet-of-things design and implementation (IoTDI). IEEE; 2016, p. 72–82. http://dx.doi.org/10.1109/IoTDI.2015.34.

[25] Yassein Muneer Bani, Shatnawi Mohammed Q, Al-zoubi Dua'. Application layer protocols for the internet of things: A survey. In: 2016 international conference on engineering & MIS (ICEMIS). IEEE; 2016, p. 1–4. http://dx.doi.org/10.1109/ICEMIS.2016.7745303.

[26] Hedi I, Špeh I, Šarabok A. IoT Network protocols comparison for the purpose of IoT constrained networks. 2017 40th international convention on information and communication technology, electronics and microelectronics, MIPRO 2017 - proceedings 2017;501–5. http://dx.doi.org/10.23919/MIPRO.2017.7973477.

Márcio Miguel Gomes is a Ph.D. student and undergraduate professor at Universidade do Vale do Rio dos Sinos, Brazil, teaching algorithms and programming languages. He obtained his master's degree in Applied Computing at Unisinos in 2015. His research interests include Distributed Computing, Edge Computing, Internet of Things, and Sensor Network.

Rodrigo da Rosa Righi is professor and researcher at Universidade do Vale do Rio dos Sinos, Brazil. Rodrigo concluded his post-doctoral studies at KAIST-Korea — under the following topics: RFID and cloud computing. His research interests include load balancing and process migration. Finally, he is a member of the IEEE and ACM.

Cristiano André da Costa is a full professor at Universidade do Vale do Rio dos Sinos, Brazil, and a researcher on productivity at CNPq. His research interests include ubiquitous, mobile, parallel and distributed computing. He obtained his Ph.D. degree in computer science from the UFRGS University, Brazil, in 2008. He is a member of the ACM, IEEE, and IADIS.

Dalvan Griebler received a Ph.D. in computer science from the Pontifical Catholic University of Rio Grande do Sul (PUCRS) and University of Pisa, Italy. He is a part-time Professor at Sociedade Educacional Três de Maio, Head of the Laboratory of Advanced Research on Cloud Computing, Postdoctoral Researcher at PUCRS, and Research Coordinator of the Parallel Applications Modeling Group.