Dynamic Collaboration of Centralized & Edge Processing for Coordinated Data Management in an IoT Paradigm

Roger Young, Sheila Fallon, Paul Jacob
Software Research Institute, Athlone Institute of Technology,
Athlone, Co Westmeath
r.young@research.ait.ie, sheilafallon@ait.ie, pjacob@ait.ie

Abstract- Over the past decade, much focus in the area of Technology has deviated towards two relatively new areas; "The Internet of Things" and "Machine Learning". Although completely separate technologies, they have one major factor in common, Data. The IoT paradigm relies on sensor devices to ingest data and gain valuable insight on their surrounding environment. Data is often considered the newest natural resource. Analysing data instantaneously can give companies a leading edge in their market. Machine learning algorithms are helping companies achieve this feat in the most efficient way possible. In this paper, we propose a governance architecture for dynamic distributed data mining, utilizing a flow based programming inspired model. We illustrate a collaborative protocol between edge devices and central controllers where computation and distribution may be driven by factors including hardware limitations, latency, or energy consumption. Our proposed architecture is evaluated in a connected vehicle use case. To demonstrate the feasibility of our work, we present two scenarios; local real-time prediction of driver alertness, and task/computation offloading based on CPU usage of the edge device.

Keywords: Distributed Data Processing, Edge Computing, Apache Nifi, Apache Minifi, Internet of Things

I. INTRODUCTION

The Internet of Things is a paradigm in which a large variety of appliances will be connected to the web. [1] Predicts 28 billion devices online by 2021. With such high connectivity to the web, much research exists to address the oncoming issues surrounding the exponential increase of data. With almost all devices assigned an ip address for the collection of data, our already strained internet will need renovation in many areas. These small sensor devices will communicate their data to a central location, often through an edge gateway device. For IoT to be a success, these "Edge Devices" must come with the capability to process and analyse data as close to source as possible. For this reason, many of the world's leading technological companies have invested heavily into research and development of edge/fog computing.

Industries, such as the automotive, have also invested into this area. This is due to estimates of a quarter billion connected vehicles on the road by 2021. With projections of connected vehicles creating 25 GB of data per hour [2], and self-driven cars creating 4000 GB of data per day, it is inconceivable to think our current infrastructure has the capability of maintaining such volumes of data. Furthermore, it may prove an unnecessary expense to store all data generated from IoT devices, while bandwidth cost over LTE is expensive. Analysing data close to source may provide the best solution for both cases.

Our evaluation focuses on a vehicle scenario, where we predict driver drowsiness in real time. Driver Drowsiness is regarded as one of the most common factors in road crashes. Although official statistics [3] report sleepiness at the wheel comprises only about 1–3% of all accidents, [4] Claims it is likely responsible for between 10-30%. Unfortunately, such figures are hard to gauge as drowsiness can go largely unreported. For this reason, alternative approaches are required to monitor driver behaviour on a continuous basis.

Both edge and centralized computing have their benefits and limitations. Edge computing allows for low latency real time processing, but may have low storage resources. On the other hand, centralized computing offers a large abundance of storage, but may not be suitable for applications demanding low latency real time processing. Therefore, an architecture that converges both paradigms offers the best of both worlds in terms of data mining large datasets in a low latency manner.

We propose a highly configurable architecture that coordinates data mining tasks between the edge and cloud containers. This is achieved through building models centrally and distributing the model to the edge devices. New models can be distributed and implemented seamlessly. Sensor data is processed and stored locally, and can be further analysed by stored functions on the edge container, or functions received from the central container. Computation and distribution may be driven by factors including latency, energy consumption, or hardware limitations. The architecture consists of a novel combination of state of the art, open source technologies. Figure 1 illustrates an IoT scenario showing cloud and edge computing.

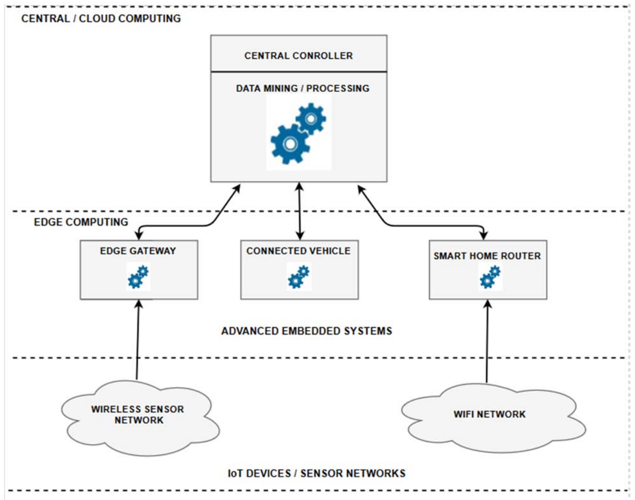


Figure 1: IoT scenario with edge and cloud computing

Our work is influenced by the Flow based programming (FBP) model. FBP can be seen as a technology where an application is constructed as a network of asynchronous processes exchanging data chunks and applying transformations to them. The creator of FBP proposed that developers spend less time thinking about the order in which things are executed (control flow), and more time focusing on the data and the transformations that are applied to it (data flow) [6]. The goal of FBP is that application development has a more natural flow to it. Although first created at IBM in the late 1960s as a software development paradigm, there has been a noticeable increase in technologies inspired by the FBP paradigm recently. [7] Discusses many of the inherent benefits with the data flow /flow-based programming paradigm, including implicit pipeline parallelism, exceptional composability, testability, inspectability and code re-use. Projects such as NoFlo [8], NodeRed [9], and Apache Nifi [10] have begun to focus on the strengths of FBP and the processing of data flows, which is a major requirement of the modern data-driven applications, thus making it a viable programming model for this oncoming paradigm shift.

This paper is organized as follows. Section II discusses related work. An overview of candidate technologies used is presented in Section III. Section IV discusses the reference architecture followed by Section V and VI which discuss the implementation architectures. A system evaluation and results are presented in Section VII. Finally, conclusions and future work are described.

II. RELATED WORK

In [11] we propose a distributed data processing architecture for edge devices in an IoT environment. Our approach focuses on a vehicular trucking use case. The traditionally centralized Storm processes such as calculating average speeds and aggregating driver errors are recreated on the edge devices using a combination of Apache MiNiFi and the user's custom-built programs. However, communication was one directional in this use case, as information was not sent from the central server to the edge devices. [12] Introduces MADAM: A Distributed Data Mining System Architecture Using Meta-Learning. MADAM is a distributed data mining architecture that combines multi-agent system and meta-learning with an objective to enhance the performance of distributed data mining system. The aim of this paper attempts to address several issues of DDM in order to be more useful across applications.

[13] Proposed a novel dynamic data flow processing platform that can dynamically change the data flow structure flexibly by extending topic-based pub/sub (TBPS) messaging method. A peer-to-peer-based data stream routing algorithm called "Locality-Aware Stream Routing (LASR) which can change the data stream destination dynamically is proposed. The authors of [14] proposed an event and clustering analytics server that acts as an interface for novel analytical IoT services. The paper introduces the OpenIoT approach to data stream analytics by use of intelligent servers running in cloud environments and edge servers for real time data acquisition and processing of sensor data. Sensor data acquired from mobile devices can be integrated into IoT platforms to enable analytics on data streams

A real-time job scheduler in Hadoop for Big Data is presented in [15]. The scheduler aims to manage cluster resources in such a way that the real time jobs will not be affected by the long running (batch jobs), and vice-versa. The case study is applied as support for Smart City applications, taxi cabs in particular. Although efficient in its design, all data is sent to the centralized scheduler for processing.

Using a similar architecture, Hortonworks demonstrated the simulation of bi-directional data communication between an on-vehicle platform and the cloud [16]. This was achieved by loading MiNiFi onto a custom Qualcomm modem located in a connected car, allowing the vehicles to transmit data to their HDF (Hortonworks Data Flow) platform [17]. The demo highlighted how to deliver critical capabilities for vehicle communication. The HDF platform could process key data such as speed and geo-location in real-time. Minifi could manage how and when to transmit much larger but less time-relevant data, (system diagnostics, etc.) This data could be batched on the vehicle and sent in bursts over known Wi-Fi locations. This is an effective solution as bandwidth over LTE is expensive.

III. TECHNOLOGY OVERVIEW

Apache NiFi [10]**,** is a data in motion technology that uses flow based processing. NiFi provides a user friendly GUI and contains over 200 processors. Each processor performs an action on the passing data. The user can create a real time dataflow by dragging Processors onto the canvas. Data flowing through the NiFi dataflow is referred to as a flowfile. Each processor can be individually configured before connecting them to the following processor. The built in NiFi processors can perform a wealth of actions such as converting data formats, adding attributes to the data, and routing data based on attributes. A collection of processors available for ingesting data from a multitude of sources including urls, ports, databases, local file systems, and external sources such as edge devices.

NiFi was created by the National Security Agency (NSA), and acquired by Hortonworks, a data analytics software company. NiFi addresses many of the technical challenges associated with IoT. NiFi adds extra security to the transportation of data with built-in support for SSL, SSH, HTTPS, encrypted content and role-based authentication/authorization and handles a diversity of datatypes as described above [11]. Apache MiNiFi [18] is a sub project that can perform almost all of the actions NiFi can. It is much more lightweight and is optimized to perform on smaller edge devices. Dataflows are created on the central NiFi server and downloaded onto the edge devices featuring MiNiFi.

Anaconda a python based Data Science platform [19] was used the build and load the models. The primary library used within Anaconda was Scikit-learn [20], a machine learning tool built on NumPy, SciPy, and matplotlib. All of the technologies used are open source and readily available. TPOT [21], a Python Automated Machine Learning (AutoML) tool that optimizes machine learning pipelines using genetic programming, was an additional package that was installed on Anaconda. TPOT can be viewed as a Data analyst assistant. It works by intelligently exploring thousands of possible pipelines to find the best one for your data. Once completed, it provides a python code to build the model with the most optimized hyper parameters.

IV. REFERENCE ARCHITECTURE

Our architecture consists of three main components:
IoT Device – The IoT devices are the small sensor devices that create the data. In most cases, these will have limited resources, and transmit data to an edge gateway device for processing.
Edge Container – The edge container represents an edge device with local processing capability. This device acts as a gateway for IoT devices. Data is ingested from the IoT devices and can be processed locally or offloaded to the central container. The edge container can also be viewed as an agent to the central container
Central container. The central container can be viewed as a cloud server with large processing and storage capacity.

As discussed, our work utilizes a FBP inspired model. The FBP model consists of three components: Black boxes, bounded buffers and information packets. Each black box in the application is an instance of a component that essentially receives some data, processes it and forwards the output to another black box. Black boxes connect to one another through ports defined by their components. A group of connected black boxes form a data flow. Bounded Buffers are the connections between the black boxes. The data that travels through the network, usually in the form of structured packets or streams of packets are referred to as information packets. They can only be owned by one black box at a time.

Figure 2 represents an illustration of the architecture. The service UI allows the user to interact with the dataflows between the edge and central container. Through the service UI, the user can seamlessly modify, in real time, the data mining processes performed on the edge and central container. This is achieved by passing functions or requests into the dataflow. These requests can be incorporated into the existing dataflows on the edge containers. Results can also be viewed through the UI.

The central container ingests data from the edge containers. Data is routed to a local database for storage, or the processing unit for analysis. The processing unit performs model building as it has access to a learning algorithm repository and the local database. The processing unit also performs data mining, in real time and/or batch. The central container acts as a coordinator for the edge devices/agents. It has the ability to send requests and information to its agents. This control data may be determined by external factors or user requests directly from the service UI.

The edge container ingests data from the IoT devices and passes it to the local processing unit. Custom programs or functions can be incorporated into the processing unit. Data analysis actions can be influenced by internal factors such as network connectivity, CPU or RAM usage, external factors such as latency, or requests received from the central controller.

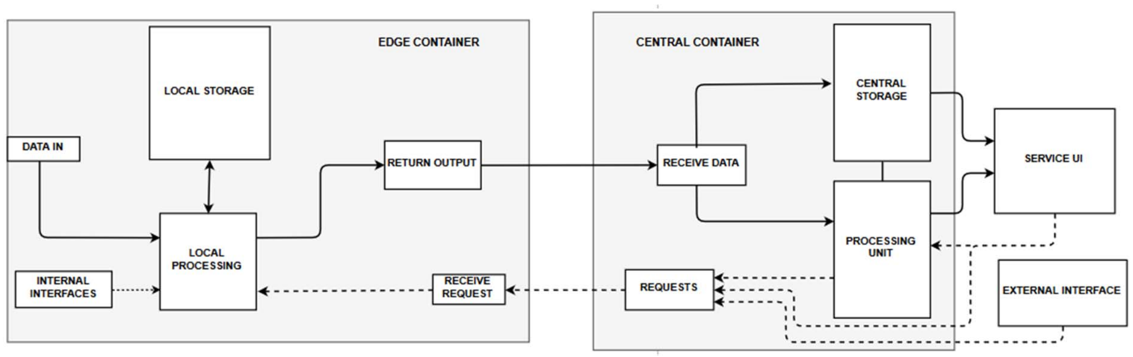


Figure 2: Reference Architecture. Edge container computation may be modified by internal or external factors.

V. IMPLEMENTATION ARCHITECTURE I

We evaluate a scenario in which data mining is implemented on a Raspberry Pi, emulating a connected vehicle. Figure 3 illustrates the dataflow on the edge container. An algorithm makes real time prediction on driver alertness. If drowsiness is predicted, an alert can be forwarded directly to the driver's phone via email. To lower communication costs, data is stored locally and can be uploaded at the end of the drivers shift, or can be sent in bursts over known Wi-Fi locations. In this scenario data is summarized in one minute time intervals and transmitted to the central container. Fig 5 shows an example of the summarized data.

The dataset used for this work was initially proposed in a Kaggle competition called "Stay Alert! The Ford Challenge" [22]. The objective was to design a classifier that detects whether the driver is alert or not, employing data acquired from over 100 participants while driving. The datasets consists of 30 features. Eight of these features are Physiological and are represented with a P, (P1, P2, P3 etc). 11 are Environmental, represented with E. 11 are Vehicular features, and represented with V. For each observation, an output "IsAlert" is labelled with 1 indicating that the driver is alert or 0 if not alert. A training and test dataset was provided. The training set was used to build the model, and the test set, which does not contain the output column "IsAlert", is used on the edge container to emulate real data transmitted from the vehicle.

TPOT was used to find and optimize the algorithm most suited for the dataset. AutoML algorithms aren't as simple as fitting one model on the dataset; they are considering multiple machine learning algorithms with multiple pre-processing steps. With the default settings, TPOT will evaluate 10,000 pipeline configurations before finishing. The user can alter these parameters to perform a quicker search if necessary. For this use case, we evaluated 500 pipeline configurations, which took 2 hours. Our system is configured to automatically distribute the TPOT model to the edge devices via Apache NiFi and Minifi.

Minifi, python and its sci-kit library were installed on a Raspberry Pi representing the connected vehicle. A Dataflow consisting of multiple processors were installed via MiNiFi. Figure 4 shows the processors used to create the dataflow on the edge container. The test dataset was placed in the Pi. A SplitText processor was configured to ingest the data one line at a time from the test dataset. From this point, each line of data is referred to as a flowfile. A ControlRate processor is configured to set the rate at which each flowfile travels through the dataflow. Here, it is set at one flowfile every 2 seconds, emulating the vehicle transmitting data in real time. An UpdateAttribute processor assigns each feature within the flowfile an attribute name. This allows the attributes to be split and routed separately if necessary.

The next step in the dataflow is to route the flowfile to the model and also the summarization process. This is achieved using a RouteText processor. Nifi will simply duplicate the flowfile when routing it to more than one destination. The processor is configured with regular expressions, flowfiles matching the regular expressions can be forwarded one direction, with the unmatching flowfile forwarded elsewhere, or dropped. In this scenario, the flowfile is duplicated and forwarded to two separate dataflows.

The first dataflow forwards the flowfiles to an ExecuteStreamCommand processor. The processor is a powerful and versatile processor that can run a custom

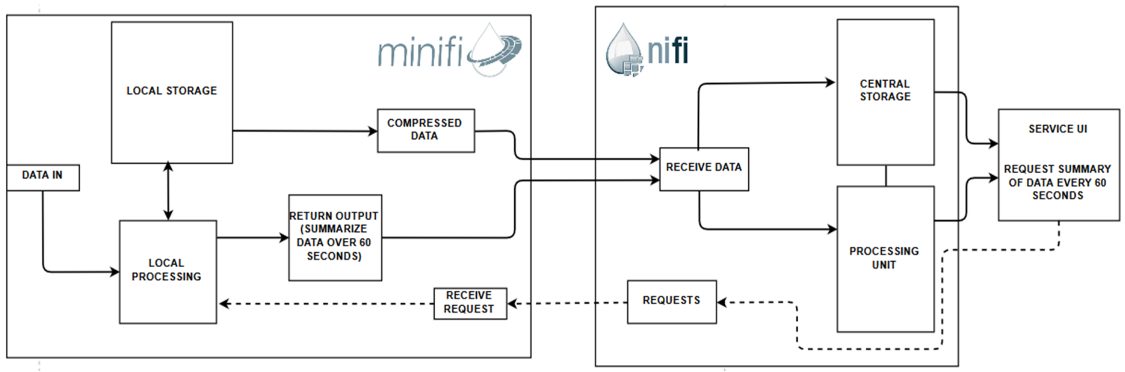


Figure 3: Implementation Architecture. Edge Container responds to Service UI request for summary of data every 60 seconds.

program within the Dataflow Here, a python code which uses the sci-kit library is called to make a prediction using the TPOT model. The output is appended to the flowfile, and stored locally. If driver drowsiness is predicted, an alert can be sent to the driver's phone and/or the fleet manager's office.

The second dataflow forwards the flowfiles to a MergeContent Processor. This processor can be configured to merge the flowfiles to a specified size. In this scenario we merged 1 minute of data. An ExecuteStreamCommand processor call another custom python code that summarizes this block of data. Figure 5 shows the output of this process. The purpose of summarizing data is to act as an update to the central server. A separate dataflow merges data together over a longer time period. This data is compressed using a Compress processer. Transmitting compressed data greatly reduces data reduction. The Nifi server ingests the compressed data from the edge devices. This is immediately decompressed and stored for further analysing.

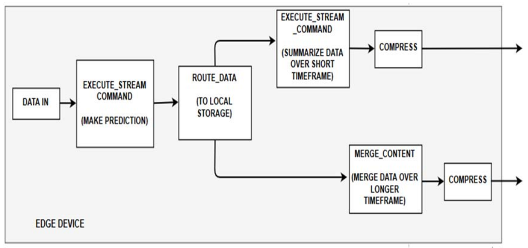
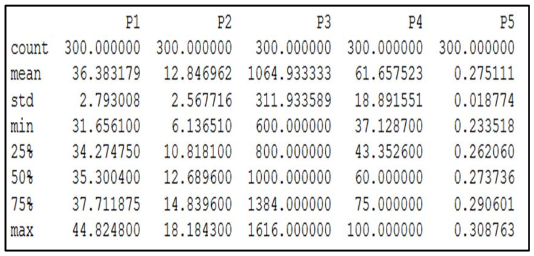

DATA IN
EXECUTE_STREAM COMMAND (MAKE PREDICTION)
ROUTE_DATA (TO LOCAL STORAGE)
EXECUTE_STREAM _COMMAND (SUMMARIZE DATA OVER SHORT TIMEFRAME)
COMPRESS
MERGE_CONTENT (MERGE DATA OVER LONGER TIMEFRAME)
COMPRESS
EDGE DEVICE

Figure 4: Dataflow on edge container



```
               P1          P2           P3          P4          P5
count  300.000000  300.000000   300.000000  300.000000  300.000000
mean    36.383179   12.846962  1064.933333   61.657523    0.275111
std      2.793008    2.567716   311.933589   18.891551    0.018774
min     31.656100    6.136510   600.000000   37.128700    0.233518
25%     34.274750   10.818100   800.000000   43.352600    0.262060
50%     35.300400   12.689600  1000.000000   60.000000    0.273736
75%     37.711875   14.839600  1384.000000   75.000000    0.290601
max     44.824800   18.184300  1616.000000  100.000000    0.308763
```

Figure 5. Example of summarized data (Count represents number of flowfiles that were summarized)

VI. IMPLEMENTATION ARCHITECTURE II

As discussed, this architecture has the capability of offloading tasks from the edge container to the cloud. Guaranteeing low-latency applications and services to the end users will be fundamental for the edge/fog computing paradigm. To provide this type of service, processing as close to the source as possible will be necessary. However, in certain scenarios, this may not be achievable due to a number of circumstances. In such a case, offloading certain tasks to the cloud may free up the already limited resources on the edge containers. Offloading may be determined by multiple factors, including, but not limited to, battery power, latency, and local environmental factors.

It will be common for edge containers to be battery powered. In such a scenario, to preserve resources it may prove beneficial to process the most time critical data locally and offload less time critical tasks when battery power is falling below a certain threshold. However, for this work we used Raspberry Pis, which were connected via AC, to emulate an edge container. Computation offloading based on battery level was not achievable. Instead, we set thresholds based on CPU usage.

We create three dataflows as shown in figure 6. Dataflow 3 is considered the most time critical data that must be processed locally at all times. Dataflow 1 and 2 represent data that can be processed locally, but not with such demand on low latency. When CPU usage rises above a specified threshold, the computations that occur on dataflow 1 and 2 are offloaded to the cloud. The cloud ingests the data from 1 and 2 and processes centrally.

Incoming sensor data is routed to the specified dataflows using a RouteText processor. Each dataflow consists of an ExecuteStreamCommand processor that calls a python script to score the incoming data off a model. The output is appended to the flowfile and stored locally. A separate system is created that monitors CPU usage frequently. For this case, a threshold of 75% CPU usage was set.

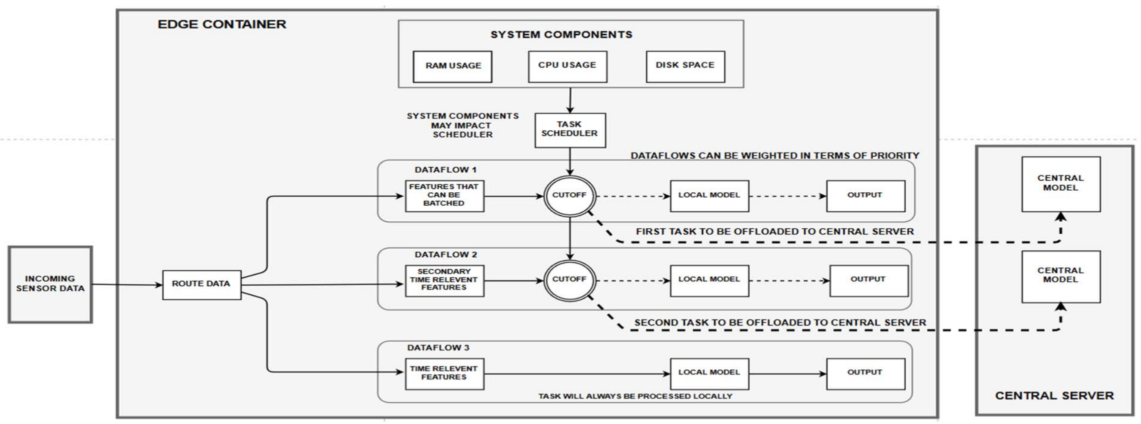

EDGE CONTAINER
SYSTEM COMPONENTS
RAM USAGE
CPU USAGE
DISK SPACE
SYSTEM COMPONENTS MAY IMPACT SCHEDULER
TASK SCHEDULER
DATAFLOWS CAN BE WEIGHTED IN TERMS OF PRIORITY
DATAFLOW 1
FEATURES THAT CAN BE BATCHED
CUTOFF
LOCAL MODEL
OUTPUT
FIRST TASK TO BE OFFLOADED TO CENTRAL SERVER
INCOMING SENSOR DATA
ROUTE DATA
DATAFLOW 2
SECONDARY TIME RELEVENT FEATURES
CUTOFF
LOCAL MODEL
OUTPUT
SECOND TASK TO BE OFFLOADED TO CENTRAL SERVER
DATAFLOW 3
TIME RELEVENT FEATURES
LOCAL MODEL
OUTPUT
TASK WILL ALWAYS BE PROCESSED LOCALLY
CENTRAL MODEL
CENTRAL MODEL
CENTRAL SERVER

Figure 6. Dataflow 3 is time critical and will be processed locally at all times. Dataflow 1&2 are offloaded to the cloud if CPU usage is high

A bash script, which represents a task scheduler, is configured to trigger an alert if CPU usage rises above the threshold for over 30 seconds. If true, dataflow 1 will cease processing locally and direct the incoming data directly to the cloud container. After this action, if CPU remains above 75%, dataflow 2 is then offloaded, (This may be true when other tasks such as system updates etc are occurring). This can be reversed when CPU returns below the threshold. Tasks/computations on the dataflows may be modified in real time to keep the CPU usage below the threshold.

VII. SYSTEM EVALUATION AND RESULTS

Implementation I Evaluation

Architectural Implementation I focused on local processing, which in turn reduced data transmission between the edge and cloud container.

The reduction of data transmission was recorded in multiple scenarios. This was achieved by increasing the control rate at which data passed through the edge device. The quantity of data produced is controlled by increasing the granularity of data production from 2 seconds, 1 second, and 500 Ms. Table 1 represents the data reduction over a 5 minute period. Full data transmission is compared against the data summary and compressed data columns combined. As shown, as velocity of data increased a significant increase in data reduction occurred. The improved performance resulted from merging the content. When data was created at a higher velocity, more content was merged in the specified time frame. However, summarized output remained the same.

Table 1: Comparison of full data transmission and summarized and compressed data

DATA INTERVALS (MS)	FULL DATA TRANSMISSION	COMPRESSED SUMMARY (1 MINUTE INTERVALS)	FULL COMPRESSED DATA	TOTAL DATA REDUCTION
2 seconds	**20.5KB**	**1.44 kb**	**2.95KB**	**78.59%**
1 second	**39.5KB**	**1.46 kb**	**6.6KB**	**79.6%**
500 ms	**79.9 KB**	**1.48 kb**	**12.5KB**	**82.5%**

This section will demonstrate the advantages of including TPOT to automatically build the models before distribution. No feature engineering was performed on the data before executing TPOT. The training dataset was split 70-30, building the model on 30% of the data and validating the model on the remaining 70%. After running TPOT for two hours on the training data, it provided a model with the most optimized hyper parameters. The chosen model was an ExtraTreeClassifier. As a comparison, five other algorithms were tested, with just their default settings. Note, one of the algorithms we tested was an ExtraTreeClassifier with its default settings. This aims to show the improvement optimizing hyper parameters make.

Table 2 represents the individual model accuracies that were tested. As shown in table 1, the automated model built by TPOT scored highest, enhancing the accuracy by 0.3% and the auc by 1.1% for this particular dataset.

Table 2: Model Scores on Validation Data

Algorithm	**Accuracy (%)**	**ROC_AUC (%)**
Logistic Regression	78.7	77.6
KNearest Neighbour	83.2	82.3
RandomForestTree Classifier	97.6	97.4
DecisionTree	96.2	96.1
ExtraTreeClassifier	97.8	96.6
TPOT Model	98.1	97.7

Overall, TPOTs inclusion to an architecture such as ours is justified due to its improved results and automated nature.

Implementation II Evaluation

System monitoring and resource management may impact decision making in terms of task resource allocation on edge containers by providing useful information such as work load and energy usage.

The efficiency of advanced embedded systems will play a major role in IoT. As previously mentioned, the edge processing for this work was carried out on a raspberry Pi. The specifications of the Pi are as follows: 1 GB of Ram and a CPU; 4× ARM Cortex-A53, 1.2GHz.

Three python scripts were running simultaneously in this scenario (one on each dataflow). Each python script was configured to ingest one line of data at a time, at intervals of 500 ms, and scored the data off a local model, as shown in figure 6.
With Apache Minifi and the three python scripts running, CPU usage was on average 77%.

In this scenario, dataflow 1 was offloaded, which lowered the overall CPU usage to 51.7%. This information allowed us to slightly adjust the manner of which we processed data in dataflow 1. For example, by modifying the script to perform a minor process, such as feature selection instead of prediction, before transmitting to a central container for further analysing, all three dataflows could be run concurrently, and below the threshold. The newly modified dataflow gave us a total average CPU usage of 66%. The graph below shows the edge containers CPU usage before and after our modification, compared to the threshold.

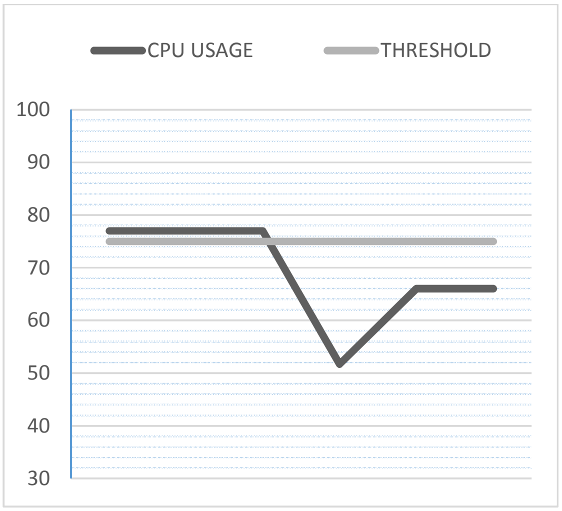


Figure 7: CPU usage vs Threshold. Drop in CPU usage occurs when dataflow 1 is offloaded to central container. Modified python script on dataflow1 keeps CPU usage below threshold

VIII. CONCLUSION

This work creates an enhanced dynamic data processing architecture in a connected vehicle environment. A reference architecture was presented and details of the components were discussed. The reference architecture utilizes the flow based programming approach. We have shown how the FBP approach allows the dataflows components to be dynamically modified. Two implementations of the architecture were evaluated. The first implementation evaluated local real-time prediction on driver alertness. We have shown that the combination of merging content and summarizing data can result in significant reductions in data transmission, upwards of 78%. The reduction in transmission exponentially increases as the granularity of data increases. The second implementation evaluated computation offloading and modification based on CPU usage. By monitoring CPU usage, we could dynamically adjust computation on the edge container to preserve hardware limitations.

There are a number of areas for future work. Task offloading based on latency between edge and central containers is an area that our proposed architecture may prove beneficial. Future work may also include implementing this architecture in a distributed density based clustering scenario. Algorithms such as DBSCAN may benefit greatly when combined with Apache Nifi and Minifi. Our architecture may be used in anomaly detection scenarios such as network security. The bi-directional relationship between NiFi and Minifi would make it possible to distribute newly detected un-signatured network threats to all network nodes in real time.

REFERENCES

[1] Gartner, "www.Gartner.com," Gartner, 26 January 2015. [Online]. Available: http://www.gartner.com/newsroom/id/2970017. [Accessed 2 May 2017].

[2] Quartz, "qz.com," Quartz, [Online]. Available: https://qz.com/344466/connected-cars-will-send-25-gigabytes-of-data-to-the-cloud-every-hour/. [Accessed 20 May 2017].

[3] Centers for Disease Control and Prevention (CD, "Drowsy Driving — 19 States and the District of Columbia, 2009–2010," MMWR Morb. Mortal. Wkly Rep., 2013.

[4] Gonçalves, M., Amici, R., Lucas, R., Åkerstedt, T., Cirignotta, F., Horne, J., ... & Peigneux, P. (2015). Sleepiness at the wheel across Europe: a survey of 19 countries. *Journal of sleep research*, *24*(3), 242-253.

[5] J. P. Morrison, "Flow-Based Programming: A new approach to application development," CreateSpace, 2010.

[6] Ivan Briano, "github.com/solettaproject/soletta/wiki/Flow-Based-Programming-Study," Soletta, [Online]. Available: https://github.com/solettaproject/soletta/wiki/Flow-Based-Programming-Study. [Accessed 2 October 2017].

[7] "FBP inspired data flow syntax," bionics.i, 16 July 2016. [Online]. Available: http://bionics.it/posts/fbp-data-flow-syntax. [Accessed 20 September 2017].

[8] H. Burgius, "Noflo–flow-based programming for javascript," 2015. [Online]. Available: http://noflojs. org.

[9] "Nodered.org," Node-RED, [Online]. Available: https://nodered.org/. [Accessed 1 October 2017].

[10] 451 Research , "Everything Flows: The value of stream processing and streaming integration," 451 Research, 2016.

[11] R. Young, S. Fallon and P. Jacob, "An Architecture for Intelligent Data Processing on IoT Devices," IEEE, Athlone, 2017.

[12] Sen, S. K., Pani, S. K., Ojha, A. C., & Dash, S. (2014). MADAM: A Distributed Data Mining System Architecture Using Meta-Learning. *IUP Journal of Information Technology*, *10*(4), 7.

[13] Y. Teranishi and e. al, "Dynamic Data Flow Processing in Edge Computing Environments," Computer Software and Applications Conference (COMPSAC), IEEE 41st Annual, vol. 1, Tokyo, 2017.

[14] Hromic, H., Le Phuoc, D., Serrano, M., Antonić, A., Žarko, I. P., Hayes, C., & Decker, S. (2015, June). Real time analysis of sensor data for the internet of things by means of clustering and event processing. In *Communications (ICC), 2015 IEEE International Conference on* (pp. 685-691). IEEE.

[15] Barbieru, C., & Pop, F. (2016, March). Soft real-time hadoop scheduler for big data processing in smart cities. In *Advanced Information Networking and Applications (AINA), 2016 IEEE 30th International Conference on* (pp. 863-870). IEEE.

[16] Guest Author, "Hortonworks.com," Hortonworks, 8 June 2016. [Online]. Available: https://hortonworks.com/blog/qualcomm-hortonworks-showcase-connected-car-platform-tu-automotive-detroit/. [Accessed 11 February 2017].

[17] "HortonWorks DataFlow," Hortonworks, 2016.

[18] nifi.apache.org, "nifi.apache.org/minifi," 18 December 2016. [Online]. Available: https://nifi.apache.org/minifi/.

[19] "https://www.continuum.io/Anaconda-Overview," Continuum Analytics, 2017. [Online]. Available: https://www.continuum.io/Anaconda-Overview. [Accessed 30 June 2017].

[20] "scikit-learn.org," Python, [Online]. Available: http://scikit-learn.org/stable/. [Accessed 10 May 2017].

[21] "rhiever.github.i," [Online]. Available: https://rhiever.github.io/tpot/. [Accessed 20 october 2017].

[22] "www.kaggle.com," Kaggle, June 2011. [Online]. Available: https://www.kaggle.com/c/stayalert#description. [Accessed 1 May 2017].