# EXTENDING THE CONTAINERS OF THE DATA PROCESSING PLATFORM ON THE EDGE OF THE NETWORK

Ľubomír URBLÍK, Sofia ČORBOVÁ, Erik KAJÁTI, Peter PAPCUN
Department of Cybernetics and Artificial Intelligence, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic, Tel. +421 55/602 2569,
E-mail: lubomir.urblik@tuke.sk

**ABSTRACT**
*This article describes the expansion of an existing data processing platform. This work aimed to analyze the current state of the platform, identify possible shortcomings, and propose new containers. These containers were later implemented and connected with the rest of the platform. The extensions described in this article provide new ways to load data from different databases or video sources, new processing tasks, and an email notification system utilizing Gmail. The results of this work provide a great stepping stone in creating the fully featured platform, which we plan on creating. It also serves as a confirmation of the ease of implementing new containers into our framework.*

***Keywords:*** *data processing, edge computing, containerization, Docker*

## 1. INTRODUCTION

Linux containers offer an easy way of deploying software applications across various devices. The application is packaged with all the required libraries, supplementary software, and files into a single package [1]. By utilizing containers, the developers can prevent issues arising from developing an application on one device and deploying it on another. Modern applications require tens to hundreds of libraries, often dependent on each other, possibly leading to issues if an incompatible version of some library is used [2]. Containers are typically used to ensure a uniform developer environment and production environment [3]. The popularity of containers rose with the onset of cloud computing. The user does not know where their application is running in the cloud. A single server can host multiple applications for multiple users, or multiple servers can host a single application. Containers allow the providers to move the applications between servers easily and quickly without affecting the user.

Edge computing differs from cloud computing in many aspects [4]. Unlike the cloud's virtually "unlimited" performance and resources, the devices are smaller and offer less performance and resources. These limitations require the developers to take a different approach to their solutions, focusing on efficiency and performance. Another issue is the heterogeneity of edge environments [5], which is still in its infancy and requires a more hands-on approach from the developers to ensure compatibility between the hardware, the OS and the software. Many of the issues arising from the immaturity of edge computing can be solved by applying containers. As long as the container is appropriately configured, it can be created on a traditional computer and then transferred to the edge device without any issues, effectively reducing the number of software dependencies to just one - the container platform.

We have decided to take a different approach to containers, inspired by nanoservices [6] and DAGs [7]. Unlike traditional virtual machines, containers can be deployed and run in seconds. Container platforms also allow for the separation of networks between the host and the containers, which provides a more secure environment. We have created a data processing framework for use in edge environments by utilising these two attributes. The application is split into multiple containers, which can be easily modified and redeployed, with each container serving a singular function. The framework also allows the application to be split between multiple devices to help ensure the best performance.

## 2. THE INITIAL SOLUTION

Due to its popularity and compatibility, we have selected Docker as our container platform, but thanks to the OCI, the framework should be compatible with other container platforms. Our application is split between multiple containers, so we selected ZMQ as our inter-container communication backbone. ZMQ is a messaging library that supports multiple patterns like pub-sub, push-pull, or client-server [8]. We have opted for the pub-sub pattern with a centralized broker. The containers communicate by sending the topic and the data to the broker, which it then forwards to the topic subscribers. We have chosen a centralized broker because of the ease of setup when sending multiple topics. In a decentralized pub-sub pattern, the publisher and subscriber must be connected directly, requiring a more extended setup. With a centralized broker, we connect all the containers to a single broker, set up the topics in each container, and let ZMQ handle the rest. The complete overview of the initial architecture is described in better detail in [9]. The solution was first aimed at use with IoT devices, such as weather stations or smart home devices.

Due to the prevalence of MQTT in IoT solutions, our initial solution could load data from the devices only via MQTT: the device and the container both connected to the Eclipse Mosquitto broker on the same topic. Whenever the device published a new message, the broker forwarded it to the container. As MQTT is not available on all devices or is not easy to set up in some cases, we have also implemented a way of loading the data via HTTP. In the initial application, only the GET method was implemented.

In this case, the device took on the server role, and the container was the client. These two methods serve a different purpose. The MQTT is aimed at scenarios where the device ensures the data is sent, either in predefined intervals or whenever a value changes. The HTTP is for scenarios where the application requires updating the data with each loop.

The first container is responsible for loading the data from the devices and transforming it into a unified form for further processing. IoT devices, store-purchased or user-built, can send data in various forms, most commonly in JSON format or as raw values, an example is shown in Listing 2. It needs to be transformed to ensure the data is then processed correctly. Listing 2 shows an example of the resulting format. The object is split into two parts - the data, which represents the data loaded from the devices, and the meta, which will contain all the metadata about the processing applied. After the data is transformed, it is sent to the ZMQ broker and enters the main and secondary processing pipelines.

```
{
    "temp" : "25"
}
```

or

```
    25
```

**Listing 1** Data transformation inputs

```
{
    "data" : {"temp" : 25},
    "meta" : []
}
```

**Listing 2** Data transformation output

The main pipeline starts with the filter, which will stop any data that is too similar to the previous value from being processed further to save space and processing power. The container requires the user to set the *precision* parameter. Whenever a new value arrives, it is compared with the last processed value and if the difference between them is less than the precision parameter, the value is not sent for further processing. This was done to prevent the solution from having to process each and every value, as all sensors have small inaccuracies in their measurements. The processing applied is saved to the *meta* field in the object, as shown in Listing 2.

After the data is transformed and filtered, it is sent to the logger container, which takes the data and saves it into a MongoDB database. We have selected a NoSQL database instead of SQL because of the variety of data structures we expect to process. Having rigid tables in our database would prove detrimental, as it would require us to create new tables for new devices to accommodate all their data fields.

The secondary pipeline is made up of a Redis Streams database, which provides an easy way of storing the raw values we get from the devices. The reason we have opted for the raw values, without any processing, is the ability to detect issues with the device for predictive maintenance. Processed values might lose their information value in cases like this. The Redis Streams is an append-only data structure where each new value is appended to the existing data. This approach is ideal for our solution as it allows us to monitor historical data together with the streaming data. After the data is saved to the database, it can be loaded by other containers.

The main strength of our solution is the ease of implementing new data processing tasks in the form of containers. Each step is developed and deployed independently, with ZMQ being the only requirement. This allows us to add new containers written in various programming languages without problems, as ZMQ supports more than 50 languages.

While the initial solution served as a great foundation, it had its drawbacks, mainly in the processing tasks offered. It did not offer enough to be usable in real-world scenarios. Therefore, we decided to expand upon it and implement new containers.

The first issue we saw with the original solution was the limited number of data sources available. MQTT and HTTP limit us only to devices with which we can connect directly. Many devices already upload their data to existing databases, which we can use as a middleman between the devices and our solution.

Another shortcoming was our focus on only tabular data. With the recent advancements in AI, and more specifically, neural networks, the focus has shifted from tabular data to video and, more recently, text. We have, therefore, decided to add a way of loading video into our framework.

The only way for a user to interact with the data was to load up the database and manually monitor the values. In practice, the user should be notified about changes in the data, such as quickly rising temperatures. To address this issue, we have added an email notification container connected to the filter container.

## 3. EXTENSION OF THE EXISTING SOLUTION

The initial solution utilized Kubernetes and Terraform, two technologies prevalent in cloud computing. Kubernetes allows us to manage our application across multiple devices easily. Terraform provides consistency when it comes to deployment. The deployment specification for our application is written inside multiple files and ensures our application will work exactly the same, no matter where it is deployed. But this approach proved a hindrance when creating new containers. Kubernetes requires at least two devices - a manager and a worker. Terraform also requires a connection to the cloud, to check and compare the state file stored in the cloud with the current state of the deployment. This is great for production deployments, but is detrimental to local development. We have therefore decided to modify the solution to utilize Docker compose. Instead of having tens of files storing the information necessary for our deployment we use only a single file, which contains all the variables and configurations of our solution. We plan

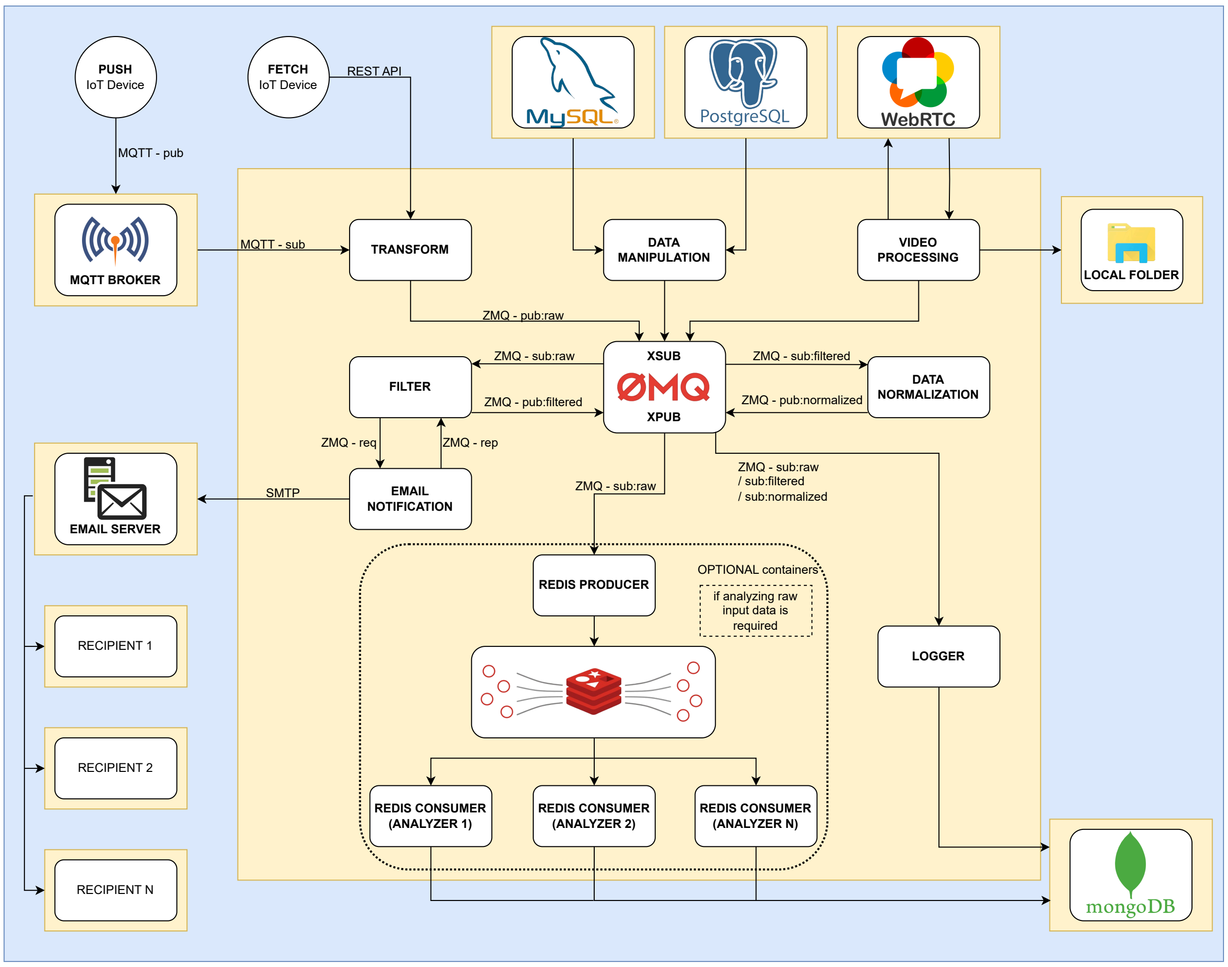

**Fig. 1** The expanded architecture of our solution

on returning back to Terraform for production deployments but when it comes to development, Docker compose proved to be a more beneficial approach. The full architecture of our solution with the new additions is shown in Fig. 1

## 3.1. Normalization

The first addition was a normalization container. Many processing tasks require the data to be normalized, transformed to a scale of 0 to 1. We have added an option to include two values, the minimum and the maximum. Any value outside of this interval is set to 0 or 1, and any values inside the interval are scaled accordingly. This step can be placed anywhere after the data transformation, giving us the flexibility of when to apply the normalization.

## 3.2. Filtering

The second addition was a new filter. IoT devices may send incorrect values, especially user-created solutions using Arduino, ESP or similar platforms. The problem may arise for any number of reasons, including a bug in the code, insufficient power, spotty connection and more. An example may include a weather station's temperature sensor that is not connected to the device properly and sometimes sends a value of 999 $^\circ$ C, which is incorrect. To filter out these values, the user can set a lower and an upper boundary, which will stop any values outside of the set interval from being processed further.

## 3.3. Notifications

The third addition was a notification service using Google Mail. This container is directly connected to the filter container as it uses the filter as a trigger. The user can set a value trigger, e.g. a temperature reaching 30 $^\circ$ C, and whenever the filter detects this value, the notification container is triggered, and an email is sent. The original idea was to implement our own email server running inside Docker. This proved problematic, as other email providers restrict unknown email servers to prevent spam. They also implement an allowlist functionality, which filters out unwanted emails but will also block our own email server. This forced us to use an established email provider, which others would not block. We have selected Google Mail, as it is a long-established email provider and should be on every other provider's allowlist. Google offers a Python library, which can be used to contact their Mail API directly. We have used an SMTP library instead because we wanted our solution not to be bound to Google Mail. SMTP is a standard protocol used for sending emails, and most providers offer an SMTP endpoint that can be used in custom applications. Our solution connects directly to the Google SMTP endpoint and sends the required parameters, which are then processed, and the email is delivered to the recipient.

### 3.4. Database connector

The next addition was a connector to SQL databases. Various relational databases do not implement the same syntax. While there are only minor differences, they might affect the functionality of our solution. To get around this issue, we used SQLAlchemy and object-relational mapper, which allows us to query relational databases without directly using SQL. Another advantage of this library is the ease of switching between different database connectors, as the connection is set up using a single string. In our solution, we have selected MySQL and PostgreSQL databases, as they are both popular and free to use. We can load historical data directly into our solution by adding databases as our new data source. We have access to the data from before it was connected to our solution and can use it to compare current values with historical trends or detect anomalies.

### 3.5. Video processing

The video processing container allows us to capture and process data from various video devices. We have selected WebRTC [10] as the main protocol used in our container. This protocol is used by real-time communication platforms and is supported in most browsers. As there is no single protocol used by all cameras we have instead focused on simplicity and ease of visualization. The container uses *rtcbot* [11] library, which is focused on Raspberry Pi.

The container connects to the camera and starts receiving the video frame by frame. When deploying this container, the user can set the framerate and the duration of each video. The program keeps count of how many frames it received and saves the recording when it crosses the selected threshold. The videos are saved using the OpenCV library in a local folder and are available for later viewing.

The main drawback of this approach is the necessity of connecting the camera directly to the device running the container. We plan on implementing other protocols in the future to allow any camera to be connected to our solution. There are existing solutions, e.g. go2rtc [12] which provide this service and we plan on modifying them to fit our needs.

## 4. TESTING AND RESULTS

Our solution was already tested, and the results were published in [9]. This paper will only contain the results of tests performed on the newly added components. All tests were performed on a desktop computer running Windows 11 Pro with Docker running inside WSL. The CPU was AMD Ryzen 5700x at stock configuration and WSL was dedicated 16 GB of 3200MHz DDR4 RAM.

The newly added database connector was tested in multiple scenarios. We tested MySQL and PostgreSQL, both running inside their own Docker Container. The tests used 4 different delays between runs - 1s, 0.5s, 0.1s, and 0.01s. We expect the most common use case to be loading a new value in the database in predefined intervals, so we created a script to select the newest value from the table and send it via ZMQ to another container. The tests were performed in Python and measured with the built-in *time* library.

### 4.1. Communication latency

The first tests focused only on the latency of the communication via ZMQ. We started a timer when the message was sent and stopped it when it arrived in the other container. The results are shown in Fig. 2. There were eight tests, 4 for MySQL and 4 for PostgreSQL. The MySQL values are always shown first, followed by PostgreSQL. The results were very similar in both cases, as was expected. The latency decreased as the interval between messages decreased, as the process was given a higher priority and/or the CPU increased its frequency to compensate for the number of messages.

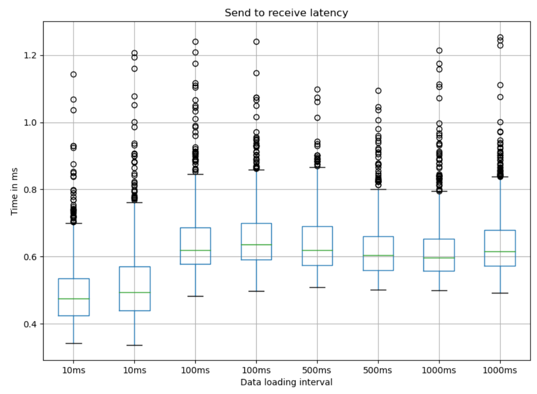

**Fig. 2** The latency of communication between two containers

### 4.2. Total latency

The second round of tests included the process of loading the data from a database and sending it to another container. The latency was measured from the start of the load process until the message was received. The results of these tests are shown in Fig. 3. As in the previous figure, MySQL results are shown first, followed by PostgreSQL. These results show a measurable difference between the database types. PostgreSQL is consistently faster by about 10ms and more consistent. The gap widens at 10ms as PostgreSQL is approximately 20ms faster than MySQL.

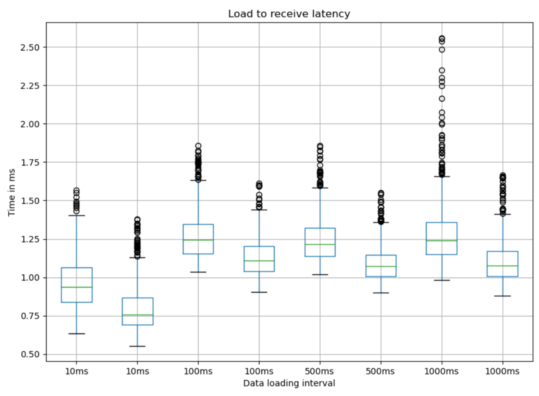

**Fig. 3** The total latency of loading, sending, and receiving the data

We have also performed a series of tests to measure the latency between sending and receiving the email notification. We have tested 4 quantities of emails - 1, 50, 100, 200. The larger numbers were tested to see if Google implements some throttling when sending multiple messages in a short duration, with 1s between each mail. The latency was measured by comparing the email metadata with the timestamp inside our program. The results are shown in Table 1. The difference between sending one email or 200 in a row is within a margin of error.

**Table 1** The median latency per one email

| Count | 1 | 50 | 100 | 200 |
|---|---|---|---|---|
| Latency | 2252ms | 2248ms | 2172ms | 2171ms |

## 5. FUTURE WORK

We want to continue expanding this platform by adding new features. Currently, we use Docker Compose files, which are harder to understand for users who have never worked with Docker before. One of our primary goals is the addition of a GUI to make the process of creating and deploying the processing pipeline easier. We are drawing inspiration from no-code platforms, which allow users to connect various parts of the application without manually setting everything up in the command line. We plan on utilizing an existing tool - Node-RED - and modifying it to our needs. The user will be able to select containers, connect them visually, and set every parameter inside this GUI. After the user sets up the pipeline, the GUI sends the entire configuration to the backend, which deploys all the necessary containers with the selected parameters.

Another goal is to implement machine learning techniques such as regression, SVM, Bayes, or Random Forest in a configurable container so it can be connected to the rest of our framework. We will also need to add new functionalities, such as aggregation or the ability to join multiple data sources. To expand on this, we plan on including an interface to show the results from our analyses, most likely Graphana, as it is flexible and customizable.

## 6. CONCLUSION

This research aimed to expand an already existing framework to offer new containerized tasks related to data processing. We have analysed the framework's current state and identified several tasks to be added. During development, we focused on the flexibility of our solution to provide customizability and prevent vendor lock-in.

The testing of our solution was focused on latency, an important criterion when developing edge computing solutions. The database connector clearly favoured the PostgreSQL database, which provided faster responses. The email notifications showed satisfying latency without any throttling in case of multiple messages in a row.

The implemented containers represent a stepping stone for other containers, which will be added later. They also highlight the flexibility of our framework, as they were added without any significant obstacles and can connect to existing containers by changing a single parameter inside them. As previously mentioned, we plan on expanding this framework to provide a customizable data processing platform which does not require any programming from the user. At the same time, it should provide a way for more inclined users to add their own containers.

## ACKNOWLEDGEMENT

This publication was supported by the APVV grant ENISaC - Edge-eNabled Intelligent Sensing and Computing (APVV-20-0247).

## REFERENCES

[1] Docker, Use containers to Build, Share and Run your applications, `https://www.docker.com/resources/what-container/`

[2] npm, Dependency Hell, `https://npm.github.io/how-npm-works-docs/theory-and-design/dependency-hell.html`

[3] Google, What are Containers? ,`https://cloud.google.com/learn/what-are-containers`

[4] IBM, What is edge computing?, `https://www.ibm.com/topics/edge-computing`

[5] Intel, Unleash IoT with Intelligent Edge Devices, `https://www.intel.com/content/www/us/en/edge-computing/edge-devices.html`

[6] KLEIN, E., Nanoservices vs. Microservices, `https://logz.io/blog/nanoservices-vs-microservices/`

[7] MALASKA, T.: Rebuilding Data Pipelines Through Modern Tools. O'Reilly Media, Inc. ISBN 9781492058168 `https://www.oreilly.com/library/view/rebuilding-reliable-data/9781492058175/`

[8] ZeroMQ `https://zeromq.org/`

[9] URBLIK, L. et al.: *A Modular Framework for Data Processing at the Edge: Design and Implementation*, Sensors **23**, No. 17 (2023) 7662

[10] WebRTC, `https://webrtc.org/`

[11] rtcbot, `https://pypi.org/project/rtcbot/`

[12] go2rtc: Ultimate camera streaming application with support RTSP, RTMP, HTTP-FLV, WebRTC, MSE, HLS, MP4, MJPEG, HomeKit, FFmpeg, etc., `https://github.com/AlexxIT/go2rtc`

Received July 2, 2024, accepted September 26, 2024

## BIOGRAPHIES

**L'ubomír Urblík** is a PhD student at the Department of Cybernetics and Artificial Intelligence (DCAI), FEEI, TUKE. His research focuses on utilization of containers in edge environments, more specifically AI and data processing.

**Sofia Čorbová** graduated (Bc) in 2024 at the Department of Cybernetics and Artificial Intelligence, FEEI, TUKE.

**Erik Kajáti** leads the Centre of Applied Cybernetics at the Department of Cybernetics and Artificial Intelligence (DCAI), FEEI, TUKE. His research delves into Industry 4.0/5.0, Human Cyber-Physical Systems (H-CPS), the Internet of Things (IoT), Edge-Enabled Computing, and the energy sector, particularly resilient and sustainable energy. His primary research focus is on the Edge-Enabled Approach for Intelligent Human-System Interoperability. Additionally, he co-founded the startup CHECkuP (Cognitive Healthcare Platform), which secured first place at the 2019 Slovak University Startup Cup and participated in the University Startup World Cup 2019. Erik is also an active IEEE and Slovak Society for Artificial Intelligence member.

**Peter Papcun** is an associate professor and the head of the Department of Cybernetics and Artificial Intelligence (DCAI), FEEI, TUKE. His research is focused on Industry 4.0, Cyber-Physical Systems (CPS) and Industrial Internet of Things (IIoT).